DEFINICIÓN DE UNA ARQUITECTURA UTILIZANDO LENGUAJE DE MODELADO UNIFICADO (UML), EN LA IMPLEMENTACIÓN DE UN LENGUAJE ESPECÍFICO DE DOMINIO EMBEBIDO (LEDE): CREACIÓN DE UN LED EMBEBIDO EN RUBY PARA EL MODELADO DE PROBLEMAS DE OPTIMIZACIÓN

ALEJANDRO RODAS VÁSQUEZ

UNIVERSIDAD TECNOLÓGICA DE PEREIRA
FACULTAD DE INGENIERÍAS
PROGRAMA DE INGENIERÍA DE SISTEMAS Y COMPUTACIÓN
PEREIRA
2015

DEFINICIÓN DE UNA ARQUITECTURA UTILIZANDO LENGUAJE DE MODELADO UNIFICADO (UML), EN LA IMPLEMENTACIÓN DE UN LENGUAJE ESPECÍFICO DE DOMINIO EMBEBIDO (LEDE): CREACIÓN DE UN LED EMBEBIDO EN RUBY PARA EL MODELADO DE PROBLEMAS DE OPTIMIZACIÓN

ALEJANDRO RODAS VÁSQUEZ

Trabajo de grado presentado como requisito para optar al título de MAGISTER EN INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

Director
Jorge Iván Ríos Patiño

UNIVERSIDAD TECNOLÓGICA DE PEREIRA
FACULTAD DE INGENIERÍAS
PROGRAMA DE INGENIERÍA DE SISTEMAS Y COMPUTACIÓN
PEREIRA
2015

NOTA DE ACEPTACIÓN

_____
_____
_____
_____
_____

FIRMA JURADO

_____

Pereira, ____ de ___ de____

CONTENIDO

LISTA DE TABLAS

LISTA DE FIGURAS

LISTA DE ANEXOS

1. PRESENTACIÓN DEL PROYECTO
1.1 INTRODUCCIÓN

La presente investigación muestra el proceso de construcción que llevó a la definición de una arquitectura utilizando el Lenguaje de Modelado Unificado (UML), para la implementación de un Lenguaje Específico de Dominio Embebido (también llamado Lenguaje Específico de Dominio Interno) el cual está orientado el modelado de problemas de optimización.

En el texto se observa las fases llevadas a cabo, presentando en primera instancia la teoría necesaria para la creación de un Lenguaje Específico de Dominio Interno y mostrando al lenguaje de programación Ruby como el seleccionado para la implementación de la arquitectura.

Del mismo modo, se evidencia el proceso evolutivo que tuvo la arquitectura como también los requerimientos que fueron surgiendo en cuanto a la necesidad de una sintaxis concisa y adaptada al dominio específico.

1.2 PLANTEAMIENTO DEL PROBLEMA

Desde el principio de los tiempos, el hombre ha utilizado señales, diagramas y dibujos para representar el mundo que lo rodea. La necesidad de representar sus ideas, lo ha conducido a un proceso de abstracción que dio como origen la creación del lenguaje hablado y escrito. En la actualidad, el hombre sigue en esta búsqueda donde las ciencias de la computación han permitido la creación de lenguajes artificiales, en especial los de programación, que ayudan a concretar dichas abstracciones, que se conciben en el análisis de un problema o situación en particular.

Cada uno de estos escenarios maneja su propio contexto el cual posee una semántica propia de su universo, es decir, cada entorno maneja su propia dinámica y términos que hacen posible la interacción dentro de este espacio, y es allí donde estos poseen una significancia válida, de modo que si se utiliza en un espacio diferente carecerían del significado que los hacen válidos.

Por ejemplo, imagine que ha ingresado a una organización dedicada a la intermediación financiera y se pide que se modele su sistema de intermediación enfocándose en las actividades de comercio y liquidación. Un paso previo a este proceso es la familiarización con los términos propios del contexto y la semántica pertinente a ellos.

Para ilustrar mejor lo anterior se presenta un problema orientado al sector financiero [4], allí se solicita la creación de este tipo de sistema:

> *Una transacción es desarrollada entre dos partes (contrapartes) y consiste en un canje de valores y divisas que está sujeto a las regulaciones del mercado en el que tiene lugar. La transacción es sólo una promesa, y tiene que ser establecida dentro un número fijo de días después de la operación. Esta fecha, conocida como la fecha de liquidación, depende de una serie de factores como el mercado específico en el que se ejecuta la transacción, el ciclo de vida de la divisa, la naturaleza de la transacción, y la fecha en la que se realizó la transacción (fecha de transacción) .*
>
> *Cada transacción tiene un valor en efectivo asociado. El valor en efectivo es la cantidad de dinero que hay que entregar a la parte que compró la divisa. Este valor en efectivo depende de cosas como el valor principal, el impuesto de timbre, y los honorarios y comisiones de corretaje, para nombrar unos pocos.*

Después de que la transacción es completada en la bolsa de valores, los detalles de la operación se introducen en la oficina de apoyo de la organización de comercio. Este proceso se llama enriquecimiento de la transacción.

El sistema calcula todos los detalles: la fecha de liquidación, impuesto sobre actividades económicas, comisión, y el valor en efectivo final.

Como se puede analizar, para lograr resolver el requerimiento (creación del sistema de intermediación) es necesario apropiarse de los conceptos propios del dominio del problema. Esto significa que, un dominio posee una gramática y semántica propia, que solamente tiene valides dentro de su contexto (*dominio del problema*).

Según [5] un modelo es una representación abstracta de un sistema y la porción del mundo que interactúa con él. Sin embargo, para construir este modelo es necesario utilizar herramientas (como UML[1]) y técnicas que permitan representar aquellos artefactos o componentes que constituyen el dominio del problema de modo que el modelo resultante pueda ser llevado al escenario llamado, *dominio de soluciones.*

Un *dominio de soluciones,* constituye el espacio donde aquellos componentes que pertenecen al dominio del problema son representados por medio de técnicas apropiadas [4]. Es decir, supongamos que ha escogido utilizar la metodología orientada a objetos, por lo tanto, las clases, objetos y métodos, conforman los principales artefactos del *dominio de soluciones* y por medio de estos se puede realizar una mejor representación de los componentes de alto nivel del dominio del problema.

Es en esta fase donde se utiliza por lo general los *lenguajes de programación de propósito general*[2], cuya característica principal radica en que su semántica y la gramática de las instrucciones que ellos poseen no están enmarcadas en la terminología y contexto de un *dominio específico*.

---

1 Unified Modeling Language (UML): Es el lenguaje de modelado de sistemas de software más conocido y utilizado en la actualidad; está respaldado por el OMG (Object Management Group). Es un lenguaje gráfico para visualizar, especificar, construir y documentar un sistema

2 Los lenguajes de propósito general, son lenguajes que pueden ser usados para varios propósitos, acceso a bases de datos, comunicación entre computadoras, comunicación entre dispositivos, captura de datos, cálculos matemáticos, diseño de imágenes o páginas, crear sistemas operativos, manejadores de bases de datos, compiladores, entre muchas otras cosas.

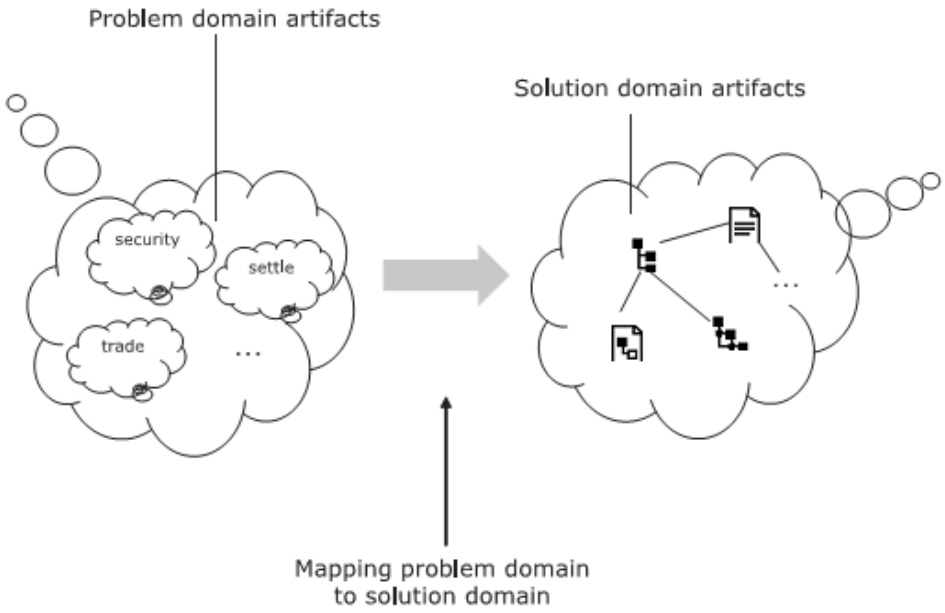


Fig. 1. Representación o mapeo entre los artefactos que pertenecen al Dominio del problema (izquierda) y el Dominio de Soluciones (derecha) [4].

Por tal motivo, cuando se llega a la situación donde las características del *dominio del problema* tiene particularidades que una herramienta genérica no puede abordar de forma efectiva[1], es necesario crear un lenguaje que sirva para el propósito requerido; esto es lo que se denomina *como Domain-Specific Language (DLS) o Lenguaje Específico de Dominio (LED)*, *donde la característica principal de los mismos, es proveer un lenguaje conciso, a medida, que sea fácil para ingenieros y expertos en el dominio de aprender, entender y aplicar para una clase específica de problema* [1].

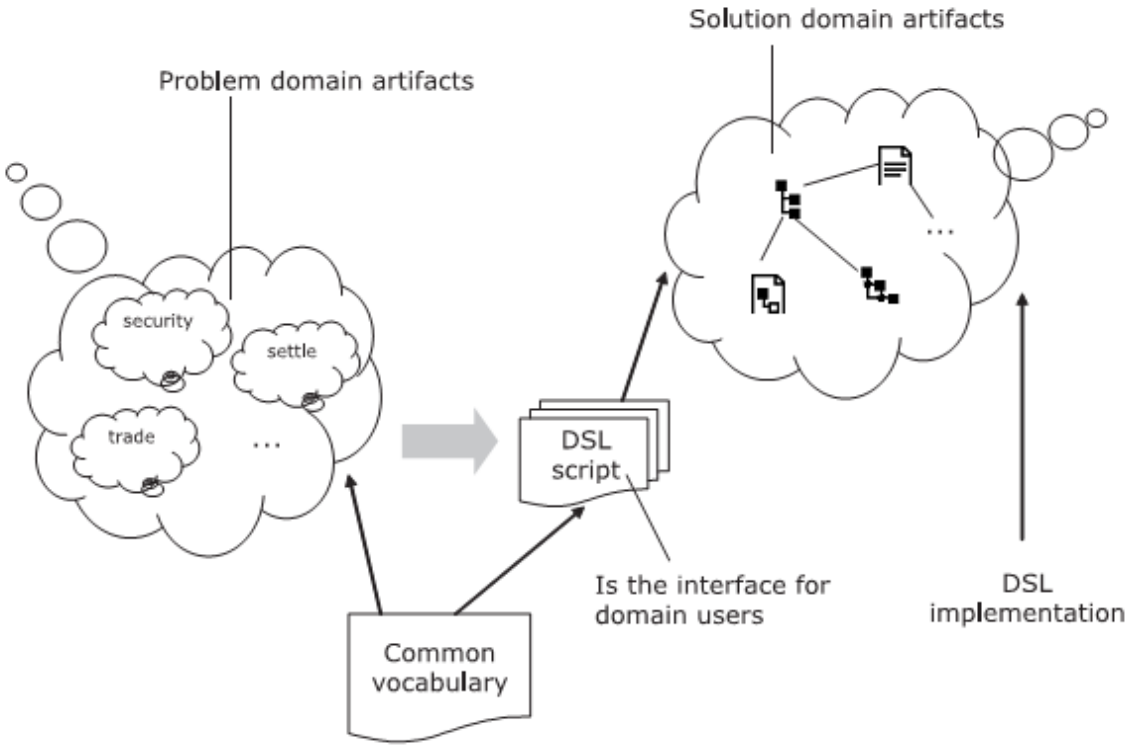


Fig. 2. Relación entre Dominio del problema, Dominio de soluciones, vocabulario común entre ellos y la creación del LED como interfaz que permite la interacción entre los dos escenarios [4].

En anterior figura, se puede observar cómo un *dominio específico* posee un vocabulario propio que permite dar forma al *dominio del problema* y simultáneamente este vocabulario puede ser expresado o manipulado por medio de un LED que sirva como interfaz entre el usuario y la máquina, de modo que todas las soluciones construidas por medio de este emplean los artefactos (clases, métodos, etc.) que están soportados a través del *dominio de soluciones.*

Ahora, una vez dada un breve pero necesaria introducción al concepto fundamental de la causa que lleva a la construcción de un LED, se puede dilucidar mejor el problema que ha llevado al inicio de esta investigación.

En efecto, durante la revisión bibliográfica se ha encontrado que existen los llamados *Lenguajes de Modelado Algebraico* (que se mencionarán más adelante), los cuales tienen como propósito servir para el modelado de problemas orientados a la optimización de funciones.

No obstante, la dificultad que tiene estos lenguajes y que es un problema evidente radica en que estos lenguajes solo pueden ser comprensibles para personas que tienen formación como programador o aquellas que estarían dispuestas a invertir tiempo considerable en aprenderlos, ya que como se muestra en la Ecuación 1.2 la forma en cómo fueron concebidos requiere un grado de conocimiento referente a la programación para lograr un manejo apropiado de estos.

Se han detectado dos problemas fundamentales de dichos lenguajes, el primero es su sintaxis y el segundo, la forma en cómo debe ser estructurado o construido el archivo que contendrá el modelo.

La sintaxis que presentan es compleja, posee caracteres que no son propios del contexto del problema y hacen que sea difícil para una persona (sin conocimiento en el área de programación) expresar los componentes que conforman el modelado un problema de optimización, como lo son la función a optimizar, sus variables y la restricciones pertinentes. Por otro lado, algunos de estos lenguajes requieren que el usuario estructure el archivo que contendrá el modelo de forma especial (Ecuación 1.2) y en el siguiente orden [39]: Declaración de datos, Variables de decisión, Función objetivo y Restricciones

Esto hace el usuario no solo deba familiarizarse con el lenguaje sino enfrentarse a problemas técnicos que trae la estructura de dicho archivo, por tanto se requiere plantear un enfoque distinto en la forma en cómo se construyen estos lenguajes.

1.3 JUSTIFICACIÓN

El planteamiento de un problema de optimización siempre se hace a través de la formulación de una función objetivo y un conjunto de restricciones que siempre se representan por un conjunto de vectores y matrices, generalmente de gran dimensión cuando se trata de problemas de la vida real.

Diseñar una función objetivo para su optimización es un proceso de abstracción donde se quiere expresar el comportamiento de cierto fenómeno por medio de sus variables representativas y algunas restricciones que están atadas a esta. Generalmente la secuencia que se sigue en dicho proceso es la siguiente [14]:

- *Formular el modelo, plantear un sistema abstracto de variables, sus objetivos y las restricciones que expresan de forma general el problema a resolver.*
- *Recolectar los datos que definan una instancia de un problema en específico.Construir una función objetivo y sus ecuaciones de restricción a partir del modelo y sus datos.*

- *Resolver el problema específico ejecutando un programa, o solver, para aplicar un algoritmo que encuentre los valores óptimos de las variables.*
- *Analizar el resultado.Perfeccionar el modelo y los datos según sea necesario.*

El tipo de problemas que poseen estas características han sido catalogados como de *programación lineal.* Como se puede anotar, el primer paso consiste en la construcción de un modelo, por lo tanto, al ser una creación salida de la abstracción de un problema se requiere un *lenguaje de modelado.* Dentro de esta categoría se encuentra AMPL, OPL, AIMMS, OptimJ, GAMS, Zimp, entre otros, los cuales se denominan como *Lenguajes de Modelado Algebraico para Optimización* o *Algebraic Modeling Languages for Optimization* para programación matemática.

Sin embargo, aunque estos cuentan con una sintaxis que permite una amplia expresividad, se necesita una persona con conocimientos en lenguajes de programación (programador) para hacer uso de ellos; enfoque muy distinto al que se propone cuando se implementa un LED.

Para dar una mejor perspectiva de este escenario se presenta el siguiente ejemplo:

Suponga [39] que se tiene un problema de producción en un fábrica que produce puertas y ventanas. Esta fábrica cuenta con tres plantas con un tiempo limitado de producción disponible.

- *Planta 1 produce el marco de metal*
- *Planta 2 produce el marco de madera*
- *Planta 3 produce vidrio y ensambla piezas*

Cada producto es fabricado en series de 200 unidades y cada serie genera una ganancia dependiendo del producto. Del mismo modo, cada serie requiere una cantidad dada de tiempo que depende de la capacidad de cada planta.

El problema radica en encontrar el número de series que se deben producir para cada producto de modo que permita maximizar los ingresos. Este problema puede ser modelado de la siguiente manera:

$$
\begin{aligned}
\text{maximize} \quad & r_d x_d + r_w x_w \\
\text{subject to} \quad & t_{d,1} x_d + t_{w,1} x_w \leq c_1 \\
& t_{d,2} x_d + t_{w,2} x_w \leq c_2 \\
& t_{d,3} x_d + t_{w,3} x_w \leq c_3 \\
& x \geq 0, y \geq 0
\end{aligned}
$$

Ecuación 1.1

Fuente: [39]

Donde r_d es la ganancia por cada producto *d*, $t_{d,i}$ es el tiempo que se necesita en la fábrica *i* para producir una serie de productos *d* y c_i es el tiempo disponible en la fábrica *i*. La serie producida es dada por x_d para cada producto *d.* Así pues, este modelo matemático puede ser representado utilizando el *lenguaje de modelado* OPL de la siguiente manera:

```
{string} Products = ...;
{string} Factories = ...;

float r[Products] = ...;
float t[Products][Factories] =  ...;
float c[Factories]  = ...;

dvar float+ x[Products];
```

Ecuación 1.2

```
maximize
  sum(p in Products) r[p] * x[p];

subject to {
  forall(i in Factories)
    sum(p in Products) t[p][i] * x[p] <= c[i];
}
```

Fuente: [39]

Como se puede observar en la Ecuación 1.2, aunque se empleen instrucciones como *maximize y subject to,* que son propias del dialecto empleado en la optimización de funciones, el resto del código es difícil de construir para una persona que no tenga conocimientos en programación (en este caso en OPL).

Por el contrario, los Lenguajes Específicos de Dominio (LED) o Domain-Specific Language (DSL) son construidos de modo que el usuario del lenguaje especifique el comportamiento que desea en términos del dominio (contexto) del problema, eliminando en la medida de lo posible aquellos caracteres (como puntos y comas, definición de tipos de datos o definición de variables) que no son parte fundamental del dominio, de modo que el lenguaje fluya por medio de expresiones naturales propias del dominio sin presentar ambigüedad en sus términos y ofreciendo la expresividad necesaria que permita el modelamiento del problema por parte de un usuario que conozca del dominio del problema pero que *no sea necesariamente un programador*.

1.4 OBJETIVO GENERAL

Definir una arquitectura utilizando el Lenguaje de Modelado Unificado (UML), en la implementación de un Lenguaje Específico de Dominio Embebido (LEDE) para el modelado de problemas de optimización.

1.5 OBJETIVOS ESPECÍFICOS

1. Escoger el Lenguaje de Propósito General que mejor se adapte según los requerimientos del proyecto para la creación de un Lenguaje Específico de Dominio Embebido.
2. Analizar los tipos de patrones de implementación para la construcción de un Lenguaje Especificación de Dominio Embebido (LEDE).
3. Seleccionar el patrón de implementación para un LEDE que mejor se adapte a las necesidades de expresividad de la sintaxis del dominio-específico.
4. Crear la arquitectura sobre la cual se sustentará el LEDE empleando UML para su diseño, de modo que permita reflejar el patrón de implementación seleccionado.
5. Demostrar el funcionamiento de la arquitectura mediante un caso práctico.

2. MARCO REFERENCIAL

Como ya se ha mencionado, un problema de optimización requiere un modelo que incluya la función objetivo, sus variables y un conjunto de restricciones. Cabe aclarar que este modelamiento es de tipo matemático he incluye los componentes ya nombrados.

La expresión algebraica que describe un modelado matemático orientado a la optimización tiene la siguiente forma:

$$\begin{array}{lll} Min_x \mid Max_x & z = cx \\ s.a & Ax = b \\ & x >= 0 \end{array}$$

Ecuación 2.1

A la hora de transmitir este modelo a una forma computacional, es necesario un lenguaje que permita plasmar el modelo descrito, es allí donde los *Lenguajes de Modelado Algebraico (LMA)* tienen su rol.

Un LMA proporciona un vínculo clave entre la concepción matemática analítica de un modelo de optimización y una compleja rutina algorítmica que busca soluciones óptimas [16]. Como se puede observar, esta definición se compone de dos partes:

- ***El Modelador:*** Está conformado por el modelo expresado desde la concepción matemática
- ***El solver[3]:*** Programa que contiene los algoritmos que resuelven los distintos problemas.

No obstante, aunque estos dos factores se complementan en realidad evolucionaron de formas separadas. Los primeros en aparecer fueron los programas que contienen los algoritmos para la resolución de problemas de optimización (*solver*), esto se remonta a 1950 [16]. Cada uno de ellos se puede dividir según el problema de programación para el que fueron diseñados como una muestra de ello se tiene a CPLEX y MINOS; CPLEX (para problemas de Programación Lineal, Programación Entera y Redes) y MINOS (para problemas de Programación Lineal, Programación No Lineal), sin embargo existen más como por ejemplo:

3 Es un programa que contiene un algoritmo para resolver problemas de optimización [16].

- CBC, Gurobi, XPRESS(para problemas de Programación Lineal Entera Mixta)
- ONOPT, MINOS, SNOPT, PATHNLP, LGO, MOSEK (para problemas de Programación No Lineal)
- DICOPT, SBB, BARON, OQNLP (Programación No Lineal Entera Mixta), etc.

Así mismo, la principal debilidad de los primeros sistemas de optimización no eran sus algoritmos, sino su interacción con el modelo. Para dar un mejor contexto, el proceso de optimización consta de cuatro fases: formulación, solución, análisis y revisión.

La fase de formulación interpreta la abstracción matemática del problema (en otras palabras, el modelo) el cual ofrece una caracterización exacta de una situación real en términos de los datos disponibles. Posteriormente, se construye un conjunto de datos que sirven para alimentar el *solver*, el cual produce los resultados que son óptimos al criterio del modelo, permitiendo que estos sean analizados y procesados.

Sin embargo, la transición que existe entre el modelo y la forma en cómo este debe ser entregado al *solver* es propenso a errores. *Mientras la forma del modelo es simbólica, general, concisa y entendible a otros modelos, la estructura del solver es contraria en todos los aspectos: es explícita, específica, extensa y conveniente para la computación [16].*

Por lo tanto, teniendo este panorama era necesario encontrar una forma que permitiera capturar el significado del modelo y al mismo tiempo, que al ser entregado al algoritmo (*solver*) no se perdiera la abstracción matemática propia del mismo. De esta manera, los LMA fueron concebidos como una manera de mitigar esta conversión.

En otras palabras, utilizando un LMA el problema abordado podría ser expresado en un lenguaje de alto nivel cercano a la expresividad humana mientras que es automáticamente traducido a los niveles requeridos por el algoritmo *(solver)*. Aunque para lograr esto fueron casi veinte años de espera desde la aparición de los propios algoritmos solucionadores, ya que *el desarrollo y la distribución de los lenguajes de modelado algebraico solo comenzaron en 1970 [14].*

Ahora observe como en la Ecuación 1.2 se tienen un ejemplo utilizando OPL. Los fragmentos de código mostrados son modelos de problemas de optimización de funciones los cuales matemáticamente expresados tienen la forma que se refleja en la Ecuación 2.1, donde se muestra la expresión de un modelo matemático para optimización.

No obstante, aunque estos lenguajes emplean palabras y símbolos que permiten realizar una descripción del problema más apegada a la expresividad humana aun poseen la limitante que para lograr un uso efectivo de ellos se debe contar con personas que tengan formación como programadores, ya que dentro de la sintaxis de los mismos existen caracteres (puntos y comas, declaración de variables, etc) que no son propios del modelado matemático del problema sino del lenguaje de programación como tal.

Por lo tanto, se necesita un enfoque que permita que las personas que conozcan del dominio del problema no tengan que emplear caracteres o palabras ajenas al contexto a modelar. Es allí donde los Lenguajes Específicos de Dominio (LED) proporcionan un camino.

Aunque los LMA pueden proporcionar un medio aceptable para realizar la optimización de una función objetivo, un LED permitiría al usuario (que no sea un programador) modelar un problema he interpretar los resultados del mismo.

Por otra parte, cabe aclarar que los LED no son una idea nueva. De hecho, antes que la programación común fuera identificada y clasificada en lenguajes de propósito general, muchos de los primeros lenguajes de programación fueron de aplicación específica [5]. Un ejemplo es APT[4] (*Automatically Programmed Tool*), el cual es *un LED para la programación de máquinas de control numérico, que se desarrolló en 1957–1958 [17].* Como una muestra de ello se tiene la siguiente tabla donde se listan algunos LED, unos más conocidos que otros.

4 El lenguaje APT provee la misma flexibilidad de expresión a los programadores de partes que los lenguajes estandar de programación proveen a los programadores de computadoras. Con APT los programadores de partes pueden definir la forma de la herramienta, tolerancia, definición geométrica, dirección del movimiento de la herramienta, posición de la herramienta en relación con el control de la superficie y comandos auxiliares[40]

Tabla 1
Listado de LED conocidos [41].

LED	Aplicación
Lex and Yacc	Análisis léxico y sintáctico
TEX, L A TEX, troff	diseño del documento
HTML, SGML[5]	marcado o etiquetado de documentos
Excel Macro Language	Hojas de calculo
SQL, LDL[6], QUEL	Bases de datos

Cabe señalar que dentro del proceso de investigación, al buscar literatura referente a los LED que muestre o teorice sobre la forma en cómo ellos deben ser construidos en realidad se encuentra muy poca. Normalmente la información que se obtiene se enmarca en un contexto general, enfocada hacia el *"qué se debe hacer"* para crear un LED y muy poca hacia el *"cómo se debe hacer".* Una muestra de esto es el libro *Domain – Specific Languages de Martin Fowler*; allí el autor revela los conceptos que se deben aplicar a nivel de programación y diseño del lenguaje de especificación, siendo este libro una guía para lograr el objetivo.

Del mismo modo, dentro de la revisión literaria se encuentran artículos como [19] y [20], que sirven como ventana para el trabajo realizado en ellos sin revelar mayores detalles. Claro está que esto no significa que los LED sean extraños en la actualidad, una muestra es la variedad que se encuentran en el área dedicada a la persistencia de datos, allí *frameworks* como *Hibernate, JQQQ, QueryDSL* o *Jequel,* los cuales buscan brindar al usuario una vía más amigable y fácil de interactuar con su motor de base de datos. También se tiene a *JQuery* y *CoffeeScript,* orientados a mejorar la experiencia con JavaScript. Por último, tal vez uno de los LEDI (Lenguaje Específico de Dominio Interno) más popular es *Ruby on Rails,* orientado a la web.

Esto demuestra que los LED pueden ocupar varias capas dentro de la infraestructura de una aplicación software, permitiéndolos clasificar según su dominio, como se muestra a continuación [18]:

5 Standard Generalized Markup Language o Lenguaje de Marcado Generalizado Estándar
6 Logical Data Language.

- *“Ingeniería del Software: productos financieros, control de la conducta y la coordinación, arquitecturas de software y bases de datos*

- *Sistemas de Software: Descripción y análisis de árboles de sintaxis abstracta, especificaciones controladores de dispositivos de vídeo, protocolos de coherencia de caché, estructuras de datos en C, y especialización de sistemas operativos.*

- *Multi-Media: Computing Web, manipulación de imágenes, animación 3D y dibujo.*

- *Diversos: Simulación, agentes móviles, control de robots, resolución de ecuaciones en derivadas parciales y diseño de hardware digital”.*

Como se puede observar, el campo para los LED es amplio y existe trabajo por hacer. Así mismo, según la investigación realizada aunque existan los LMA estos no cuentan con la expresividad necesaria que permita a una persona que no sea un programador, modelar el dominio del problema a optimizar, por lo tanto la creación de un LED enfocado en esta área brinda grandes posibilidades.

3. MARCO TEÓRICO

3.1 Diseño Basado en Modelos

Una de las principales dificultades en la construcción de software es determinar qué metodología de desarrollo es la más apropiada y se ajusta mejor a las necesidades del proyecto. En esta sección se analiza la metodología llamada *Diseño Basado en Modelos (Domain Driven Design)* la cual ha sido la seleccionada para realizar el análisis y diseño de la arquitectura sobre la cual se basará el LEDI en construcción.

Como anteriormente se ha mencionado, un LED *(Lenguaje Específico de Dominio)* es un lenguaje creado para modelar y resolver problemas que están enmarcados en un dominio específico, por lo tanto, este debe poseer dentro de su estructura sintáctica y semántica, la significancia de los términos y propiedades que son aplicados sólo al dominio del problema para el que fue creado. De esta manera, un LED debe soportarse en un *Modelo de Dominio* que refleje el contexto para el cual ha sido diseñado, siendo así el *Diseño Basado en Modelos (Domain Driven Design)* la metodología que puede brindar un aporte significativo en la creación del mismo.

Es importante hacer énfasis que el *análisis del dominio* y el *diseño* son críticas para el desarrollo del software. El *análisis del dominio del software* consiste en *la identificación, análisis y especificación de los requerimientos*, los cuales pueden obtenerse a través de diferentes modelos como lo son los *Modelos basados en el escenario, Modelos de clase, Modelos de comportamiento o Modelos de flujo* [24]. De esta manera la etapa de *diseño* puede ser alimentada por alguno de estos modelos dando como origen un producto o sistema terminado.

No obstante, uno de los problemas más comunes que se encuentra en la etapa de *especificación de los requerimientos* radica en no poder obtener un cierto tipo de *conocimiento* o información vital, ya sea porque no se lograron realizar las entrevistas suficientes con los *experto del dominio* o porque el equipo desarrollador al ser nuevo en el tema, no tenía la experticia requerida para obtener los Modelos apropiados. Esto indudablemente repercutirá en el *diseño de la arquitectura,* puesto que es en ella donde se define la relación entre los elementos principales de la estructura del software, los estilos y los patrones de diseño de la misma. Por tal razón, ese “vacío” en el *conocimiento* debe ser obtenido en alguna de las fases del desarrollo, que no son las indicadas para tal fin como puede ser en la mitad de la creación de una funcionalidad del sistema.

De esta forma, el enfoque que propone el *Diseño Basado en Modelos (Model Driven Desing)* ayuda a crear un solo modelo que captura al mismo tiempo, tanto los detalles técnicos que se necesitan para crear el *diseño de la arquitectura* como también el *conocimiento del dominio* que se debe adquirir en la fase de *análisis* del mismo.

3.2 EL DISEÑO BASADO EN MODELOS COMO HERRAMIENTA PARA LA CREACIÓN DE LENGUAJES ESPECÍFICOS DE DOMINIO

En contraste con lo ya mencionado, la creación de un Modelo empleando el *Diseño Basado en Modelos* consiste en lograr la cohesión entre un Modelo de Análisis y su Diseño. Sin embargo, la construcción de este Modelo no se limita a una simple entrevista con el *experto del dominio* y el desarrollo de un *diseño* que lo represente. A continuación se presentan una serie de pasos que según [13], permiten lograr la construcción un *Modelo* efectivo utilizando esta metodología. Del mismo modo, aunque estos pasos se han identificados de forma separada, ellos son aplicados de forma simultánea dinamizando la construcción del Modelo.

Refinamiento del modelo

Un *modelo* es básicamente el compendio del *conocimiento* que poseen los *expertos del dominio*, donde este puede provenir de una serie de fuentes como lo pueden ser los propios expertos, los usuarios existentes del sistema o de la experiencia previa del equipo técnico con un sistema heredado relacionado, tal como lo dicta el enfoque tradicional de la ingeniería de software. Para lograr este *modelo* es de vital importancia que exista una interacción entre el equipo de desarrollo y los *expertos del dominio*, puesto que es por medio de esta interacción que se plantean y se discuten cuáles son los conceptos que aportan un valor significativo al modelo y que por consiguiente tendrán relevancia dentro del mismo; entiéndase como “conceptos” a aquellos términos que juegan un papel importante dentro de la lógica del *dominio*, como lo pueden ser la propia sintaxis de las palabras, las relaciones que puede existir entre los términos y todas aquella asociaciones de índole semánticas, sintáctico y lógico.

De esta forma, teniendo en consideración estos factores, el *modelo* empieza a tomar forma, creándose así un conocimiento compartido entre los dos actores (*expertos y desarrolladores*), el cual permite ir desarrollado un esquema que es comprensible para ambos grupos, de modo que al ir mejorando la calidad de las interacciones, se empieza a desarrollar un constante *refinamiento o evolución* del *Modelo de Dominio*, forzando a los desarrolladores a aprender principios importantes del *negocio*, en lugar de producir mecánicamente funciones de código y del mismo modo permite a los *expertos* redefinir su propio conocimiento sobre el área comprendiendo mejor la lógica del negocio.

De esta forma, el *modelo* es creado de forma tal que empieza a ser lo suficientemente riguroso como construir una aplicación fácil de implementar y comprensible, no solo desde el punto de vista de un diagrama sino desde el código en sí.

Crear un lenguaje basado en el modelo

Al inicio de cualquier proyecto de software, una de las dificultades más grandes (aparte de las técnicas) es lograr la correcta comunicación entre los *expertos del dominio* y los *desarrolladores.* Cada uno de estos actores utiliza un vocabulario diferente; los *expertos* manipulan perfectamente los conceptos y términos propios del *dominio*, pero carecen de versatilidad en el lenguaje técnico en el desarrollo de software, por otro lado los *desarrolladores* muestran un panorama opuesto, son buenos en los asuntos técnicos pero necesitan tiempo para familiarizarse con el *dominio* en el que se desenvuelven los *expertos.*

De tal forma, es imperativo lograr un lenguaje en común que permita no solo comunicar a *expertos y desarrolladores*, sino que también permita la comunicación entre estos últimos. Uno de los problemas que más afectan el proceso de desarrollo de software, es cuando cada desarrollador posee su propia interpretación del *dominio del problema*, de modo que las abstracciones que realizan sobre el *dominio* son creadas para soportan solamente su diseño técnico pero no son aplicables al *conocimiento* que los *expertos* tratan de transmitir.

En consecuencia, un proyecto enfrenta serios problemas cuando su lenguaje se encuentra fragmentado. Los expertos del dominio utilizan su propia terminología mientras los miembros del equipo técnico tienen su propio lenguaje para discutir sobre *dominio* en términos de diseño. Sin embargo, este obstáculo puede ser sorteado mediante la construcción de un *modelo basado en el lenguaje.*

Un *modelo basado en el lenguaje* es el mismo *Modelo de Dominio* sin embargo es llamado de esta forma para acentuar la importancia que tiene poder crear un lenguaje en común. Este modelo [13] debe proveer no solo el lenguaje para la comunicación entre los *desarrolladores* y *expertos del dominio*, sino también entre los mismos *expertos,* de modo que puedan expresarse al momento de hablar sobre los requerimientos, planes de desarrollo, y las funcionalidades.

No obstante, su construcción es un proceso gradual que se *refina* en cada iteración, allí se deben discutir aquellos términos que incorporan dentro de su significado las reglas propias del negocio y que por lo tanto son explícitas en el modelo.

Este proceso se puede desarrollar mediante una *lluvia de ideas y experimentación*, surgiendo términos y palabras que son desechas o replanteadas, todo esto con el propósito de enriquecer *semánticamente* el modelo.

En consecuencia, no solo la sintaxis cambiará sino que estas modificaciones se proyectaran en el aspecto técnico, puesto que dichos cambios en el lenguaje son reconocidos como cambios en el *Modelo de Dominio* y llevan al equipo a actualizar y renombrar el diagrama de clases y los métodos en el código, o inclusive a modificar el comportamiento cuando el significado del término cambia.

Acoplar el modelo y la implementación

Una de las características predominantes que posee el *Diseño Basado en Modelos* consiste en que este permite conjugar las etapas de *análisis* y *diseño* existentes en cualquier proceso en la elaboración de software.

Tradicionalmente el equipo encargado de la creación del *modelo de análisis* solo se concentra en el *dominio del negocio*, recabando información sobre este y los requerimientos del cliente sin considerar en ningún momento el papel que esta información jugará en la etapa de *diseño* e *implementación*. Por consiguiente, al no considerar este hecho conlleva que la información recabada por los analistas no contenga los componentes necesarios que permitan construir un *diseño* que sea útil para la implementación puesto que los conceptos que permitirían una correcta codificación no están presentes.

De esta forma, el modelo de análisis se queda corto alcanzado su principal tarea que es entender el *dominio*, puesto que los hallazgos cruciales (relaciones entre términos, conceptos significativos, términos propios del dominio, etc) emergerían durante el *diseño* y la *implementación*. Es decir, es en estas etapas donde los desarrolladores se encuentran con los *vacíos* que no permiten plasmar el *modelo de análisis* mediante la codificación. Así que el *diseño* no reflejaría el *modelo de domino* y este modelo tendría poco valor colocando en entredicho la exactitud del software.

Por tanto, la fase de análisis no solo debe preocuparse por los aspectos generales del dominio y los requerimientos del cliente sino que debe enfocarse en la captura de los conceptos fundamentales del dominio de una forma comprensible y expresiva. Del mismo modo, la fase de *diseño* no tendría que ser vista como la etapa donde exclusivamente se construyen diagramas, sino como la fase donde se especificarán una serie de componentes que serán construidos por medio de las herramientas de programación (lenguajes de programación) empleadas en el proyecto y que toman como base el *Modelo de Domino.*

En conclusión, el emplear *Diseño Basado en Modelos* no solo se lograr crear el diseño del sistema desde un punto de vista técnico sino que al ser aplicado apropiadamente captura el conocimiento del dominio (que es construido conjuntamente entre expertos del dominio y desarrolladores) para ser plasmado posteriormente a través de código que expresará dicho modelo de forma efectiva.

3.2.1 Modelo Semántico

Al crear un LED el principal objetivo es construirlo con la expresividad suficiente que permita al usuario comunicarse en términos del dominio, lo que significa que el lenguaje se convierte en la interfaz entre el usuario y el *Modelo de Dominio* que sustenta el propio lenguaje. De modo que un *Modelo de Dominio* que es construido siguiendo las premisas que dicta el *Diseño Basado en Modelos* incorpora intrínsecamente lo que ha sido llamado por Martin Fowler [6] un *Modelo Semántico.*

Anteriormente, en el apartado *Crear un lenguaje basado en el modelo* se hizo hincapié en la necesidad de crear un lenguaje común entre los *expertos del dominio* y los *desarrolladores*, donde este sería enriquecido semánticamente con términos y definiciones creados entre ellos mismos. Así pues, es en este punto donde la semántica entra a formar parte del modelo.

Aunque el *Modelo Semántico* hace parte del *Modelo de Dominio*, no quiere decir que es un sinónimo de este, sino que es un subconjunto del *Modelo de Dominio*, esto quiere decir que los elementos que forman el *Modelo Semántico* también son parte del *Modelo de Dominio* de manera que conforman un unidad integral de un área en particular que puede ser probada independientemente para verificar si la misma produce una correcta creación semántica, de forma que para lograr esto no es necesario emplear la totalidad del *Modelo de Dominio*.

Así pues, el *Modelo Semántico* debe ser diseñado alrededor de la finalidad del LED; en el caso de esta investigación el propósito final es lograr que el LED permita realizar el modelado de un problema de optimización, en otros casos como por ejemplo para una máquina de estados, el objetivo final puede ser controlar el comportamiento de esta máquina mediante el lenguaje.

Cabe aclarar que el *Modelo de Dominio* se utiliza para referirse a un modelo de objetos (hablando en términos de Programación Orientada a Objetos), mientras que el *Modelo Semántico* pueden ser los datos por sí solos. Un buen ejemplo de esto se puede observar cuando se emplea un ORM (*Object Role Modeling*[7]) dentro de una aplicación software.

7 Mapeo Objeto-Relacional

Un ORM es principalmente utilizado para el modelado y consulta de sistemas de información a nivel conceptual (Modelo Entidad/Relación) y para el mapeo entre este nivel y el nivel lógico (Modelo de clases) [21].

En un ORM cada tabla del modelo E/R toma el nombre de e*ntidad,* que a su vez viene a ser un objeto (dentro de Modelo de Clases). Esta *entidad* debe ser identificada por una referencia bien definida en el esquema (llave primaria), la cual es usada para referirse a esta. Por ejemplo, un empleado es identificado por su número de seguridad social. De este modo, las relaciones (uno a uno, uno a muchos, etc) que existen dentro del nivel conceptual y que son propias de un modelo E/R, son interpretados a su equivalente utilizando los conceptos aplicados en el paradigma orientado a objetos (asociaciones, agregación, composición, herencia, etc). En otras palabras, el ORM se encarga de acoplar los datos que existen entre un modelo de objetos y la base de datos relacional que el modelo representa**.**

De esta forma, el LED es utilizado para describir la relación existente entre las *entidades,* es decir los objetos relacionales, dando como resultado el *Modelo Semántico* que consiste en el mapeo de los datos y por consiguiente su manipulación; a diferencia del *Modelo de Dominio* que como ya se ha mencionado es el modelo de objetos y el objetivo del mapeo.

Por otro lado, un concepto que ayuda a fundamentar una apropiada creación de un *Modelo Semántico* desde su inicio es el introducido por Kent Bench en [25], el cual es llamado *Comportamiento Específico de la Instancia* (*Instance-Specific Behavior*). Básicamente este menciona que cuando la lógica de un objeto es completamente determinada por su clase, los desarrolladores que observen el código de la clase pueden conocer qué es exactamente lo que pasa; por consiguiente, el objeto que proviene de esa clase posee un *comportamiento* y una *semántica* propia que puede ser interpretada con solo analizar el código de la clase a la que pertenece.

No obstante, la aplicación de este concepto puede parecer obvia, sin embargo las malas prácticas en el desarrollo de software hacen que se encuentren clases mal diseñadas que al instanciar un objeto, la constitución interna de este presenta diversos *comportamientos* que una vez analizado el código se encuentra que estos pertenece a otras clases que están relacionadas con el objeto, de modo que el comportamiento del mismo está disperso por toda la aplicación, donde el único modo de saber qué es lo que pasa sería analizando el flujo de datos para entender cómo se comporta el objeto en particular, lo cual genera una total pérdida del *Modelo Semántico* y del *Modelo de Dominio.*

4. C4: CONTEXTO, CONTENEDORES, COMPONENTES Y CLASES

En esta instancia del documento se ha hecho hincapié en la importancia que requiere construir un *Modelo del Dominio* efectivo que comunique el conocimiento del Dominio, donde para lograr esto se hace uso del *Diseño Basado en Modelos.* Sin embargo, aun no se ha profundizado en alguna técnica o metodología que permita no solo plasmar dicho modelo sino construirlo. En este capítulo se hace referencia a la metodología llamada C4, la cual describe cómo a través de diferentes diagramas o vista se puede describir un sistema (en este caso una arquitectura) permitiendo hacer una construcción estructurada de este.

Anteriormente ya se ha mencionado que al crear un *modelo que esta enlazado con la implementación* se tiene un referente del cómo se tendría que plantear la construcción de este. Sin embargo, es allí donde intervienen los factores técnicos que están inmersos en la construcción de un sistema que ha de ser codificado, donde los posibles inconveniente serán sorteados mediante un diseño efectivo.

No obstante, a medida que se construye el sistema es imperativo comunicar y visualizar la evolución de la arquitectura del mismo. De esta forma, UML se presenta como un medio de comunicación estandarizado que permite obtener este fin.

Como es sabido UML presenta una serie de diagramas que permiten observar el sistema dinámica y estáticamente [26]. De modo que para analizar la forma o estructura que va tomando la arquitectura del sistema a medida que va evolucionando es conveniente emplear los diagramas pertenecientes al área estructural (tabla 2) particularmente los diagramas de clases, componentes y componentes.

Tabla 2
Vistas y diagramas de UML [26].

Área	***Vista***	***Diagramas***	***Conceptos Principales***
Estructural	Estática	Clases	Clase, asociaciones, generalización, interfaz
	Casos de Uso	Casos de Uso	Caso de uso, actor, asociación, extensión, inclusión, generalización de casos de uso
	Implementación Despliegue	Componentes	Componente, interfaz, dependencia, realización
		Despliegue	Nodo, componente, dependencia, localización
Dinámica	Máquina de Estados	Estados	Estado, actividad, transición, acción
	Actividad	Actividad	Estado, actividad, transición de terminación, división, unión
	Interacción	Secuencia	Interacción, objeto,mensaje, Activación
		Colaboración	Colaboración, interacción, rol de colaboración, mensaje
Gestión del Modelo	Gestión del Modelo	Clases	Paquete, subsistema, modelo

Por otro lado, la sola escogencia de este enfoque en particular no garantiza que se alcance automáticamente un diseño que comunique efectivamente la arquitectura del sistema, así que se necesita encontrar una metodología que permita alcanzar este objetivo y a la vez permita construir diagramas que sean sencillos y sobre todo que expresen y transmitan la realidad del sistema.

Así pues, Simon Brown [27] presenta su metodología llamada **C4**, la cual propone visualizar la estructura de una arquitectura de software de forma simple, reflejada como una serie de bloques (Fig. 3) donde cada uno representa un nivel de abstracción dentro de la jerarquía.

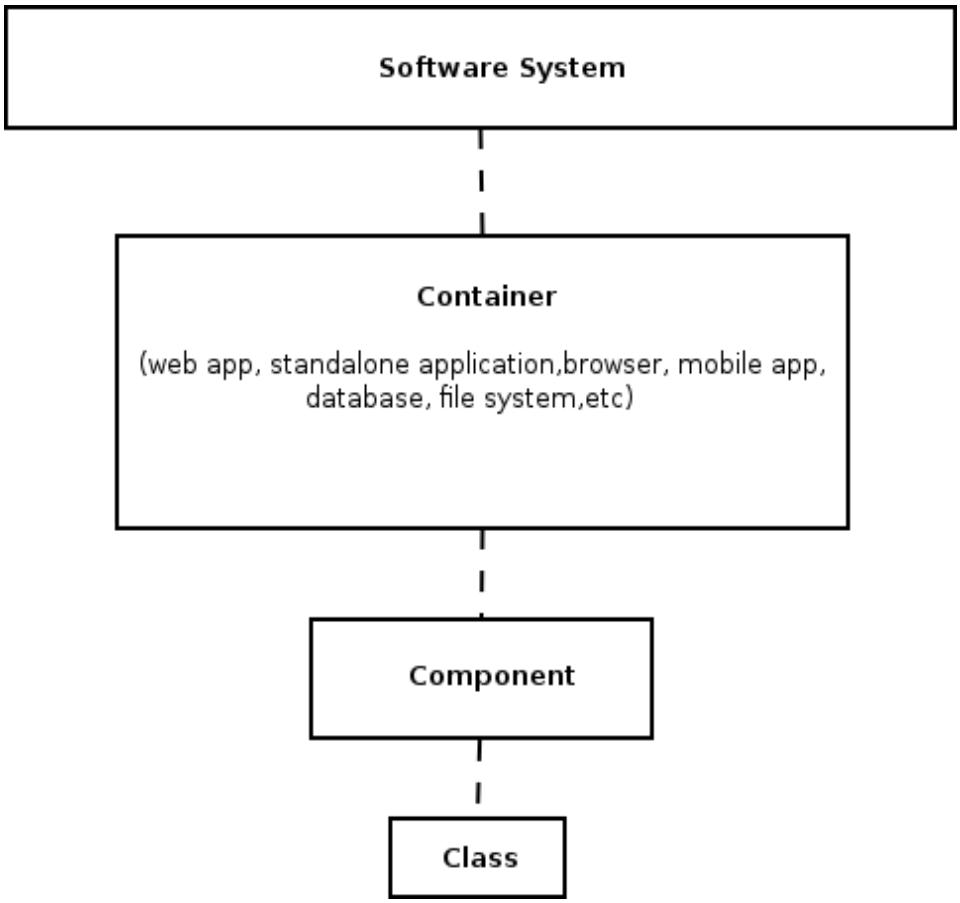


Fig. 3. Estructura jerárquica presentada un modelo simple para la construcción de una arquitectura [27].

En otras palabras, esta metodología propone que [27] un *sistema de software está hecho por un número de contenedores, que a su vez se componen de una serie de componentes y estos a su vez son implementados por una o más clases*.

A continuación se presenta la definición de cada uno de los niveles.

- ***Sistema de Software (Software System)***: Es el nivel más alto en la abstracción. En este nivel se observa al sistema en su forma general, de modo que se deben mostrar los actores que interactúan con el mismo, ya sean usuarios u otros sistemas los que interactúan este, los detalles técnicos no son importantes ya que lo que se pretende es mostrar un panorama general.

- ***Contenedores (Container)*:** Un diagrama de contenedor muestra las decisiones tecnológicas a un alto nivel, muestra cómo las responsabilidades son distribuidas a través de ellos y cómo los contenedores se comunican. Para la presente investigación se utilizó el *diagrama de paquetes* para representar este bloque, de modo que se pudiera identificar la estructura del sistema desde una perspectiva de capas como la que puede ofrecer este tipo de diagrama.

- ***Componentes (Component)*:** Por cada contenedor, un diagrama de componentes permite observar los componentes lógicos claves y sus relaciones. Para la presente investigación se empleó el *diagrama de componentes* para mostrar esta fase del modelo C4, a través de este tipo de diagramas se mostró cómo cada uno de los paquetes que conforman el bloque de contenedores es conformado por componentes que desarrollan la implementación de una determinada funcionalidad.

- ***Clases (Classes)*:** Es un nivel opcional, es utilizado para explicar cómo un patrón o componente en particular será implementado. Para la investigación fue necesario llegar hasta este nivel, puesto que para lograr la implementación del *Modelo de Dominio* que sustenta el LEDI se necesitó emplear el diagrama de clases que soportan el mismo.

De manera que al seguir el enfoque propuesto por esta metodología se tiene una forma simple de mostrar una jerarquía lógica basado en bloques que puede ser usada para modelar la mayoría de los sistemas de software de forma ordenada.

5. RUBY COMO OPCIÓN PARA CREAR UN LENGUAJE ESPECÍFICO DE DOMINIO INTERNO

Al iniciar la construcción de un LEDI se debe determinar cuál sería el *Lenguaje de Propósito General* más apropiado para actuar como *lenguaje anfitrión*. Generalmente, los lenguajes a considerar son los que habitualmente el desarrollador emplea en su actividad, ya sean *lenguajes de tipado estático* (como Java, C++, C#, C, etc.) o *lenguajes de tipado dinámico* (como PHP, JavaScript, Grooby, Phyton, Perl, Ruby, etc.)

Sin embargo, no se puede olvidar que la piedra angular que sustenta a un LED (en este caso un *Lenguajes Específicos de Dominio Interno*) es su *Modelo de Dominio*.

Así pues, la disertación que genera la escogencia de un *lenguaje anfitrión* debe estar basada en los atributos o propiedades del mismo de manera que pueda brindarle al LEDI la expresividad suficiente en su sintaxis, y al mismo tiempo permita enlazar el *Modelo de Dominio* con la tecnología que lo implementara.

Por otro lado, como ya se ha señalado el *Modelo de Dominio* albergar conocimiento del negocio, de modo que está sujeto a la evolución del mismo, por lo tanto se necesitará un *lenguaje anfitrión* que sea adaptable a esta dinámica.

Así pues, en esta sección se exponen inicialmente los conceptos de *Bloques, Proc y lambdas.* Dichos conceptos son fundamentales cuando se desea emplear la *Reflexión y la Metaprogramación* (que serán explicados más adelante) en la creación del LEDI. Cabe mencionar que estos atributos o características han influenciado notablemente en la arquitecta en construcción y han convertido a lenguaje de programación Ruby en el escogido para realizar en la presente investigación.

5.1 BLOQUES (BLOCKS), PROCS Y LAMBDAS

5.1.1 Bloques (Blocks)

Una de las características predominantes en Ruby es el uso de los denominados *Bloques*, los cuales son [22] *estructuras sintácticas* que contienen instrucciones (fragmentos de código) que pueden ser asociadas a la invocación de un método cualquiera que generalmente posee parámetros.

Los *Bloques* son delimitados con llaves ({ }) o también utilizando las palabras reservadas *do* y *end,* como se observa en el código 5.1 y 5.2, respectivamente. Allí se puede notar que la instrucción que conforma el bloque permite imprimir los números que se encuentran entre uno y diez, como lo define la instrucción *1.upto(10)*.

```
1.upto(10)  {|x|puts x}
```

(Código 5.1)

```
1.upto(10) do |x|
   puts x
end
```

(Código 5.2)

Fuente: [22]

Al asociarse un *Bloque* con un método, este puede hacer uso de las instrucciones que conforman tal *Bloque* invocándolo dentro de la definición del método por medio de la palabra reservada *yield.* Del mismo modo, a través de este medio se pueden pasar parámetros al *Bloque* para que este ejecute la acción programada. Este atributo al principio puede parecer superficial, ya que una formación tradicional en programación dicta que si se desea agregar alguna funcionalidad a un método simplemente se llama a otro método que preste la funcionalidad requerida. Sin embargo, al utilizar *Bloques* se está creando fragmentos de código que pueden *son identificados como objetos* (cuales son llamados *Procs y lambdas* y se verán más adelante), por lo tanto estos objetos pueden ser analizados y manipulados a través de programación.

En (código 5.3) se muestra cómo se puede invocar un Bloque dentro de un método. Básicamente se tiene un método llamado *sequence2* que recibe tres parámetros y tiene asociado un Bloque (encerrado entre llaves ({}), como ya se mencionó), cuya función es simplemente imprimir un resultado.

En la implementación de la función se observa la generación de una secuencia de *n* números, donde los parámetros recibidos por la función serán multiplicados por la expresión *m*i+c*; el resultado de esta operación posteriormente será mostrado utilizando la instrucción *puts* (propia de Ruby) que contiene el Bloque.

```
def sequence2(n, m, c)
    i = 0
    while(i < n)
        yield  i * m + c
        i+ = 1
    end
end

sequence2(5, 2, 2)  {|x| puts x}   #Imprime los números 2, 4, 6, 8, 10
```

(Código 5.3)

Fuente: [22]

Observe cómo una vez se ingresa al ciclo ***while*** el flujo es cedido al *Bloque* asociado por medio de la instrucción ***yield***, este muestra el resultado de la operación *m*i+c* y posteriormente el mismo flujo es retornado al método.

Por otra parte, como es costumbre en Ruby existe otra manera de invocar un *Bloque* la cual permite tener más control sobre el mismo como se muestra en (código 5.4).

```
def sequence4(n, m, c, &b)
    i = 0
    while(i < n)
        b.call(i * m + c)
        i+ = 1
    end
end

sequence4(5, 2, 2)  {|x| puts x}   #Imprime los números 2, 4, 6, 8, 10
```

(Código 5.4)

Fuente: [22]

Es importante observar que en la definición de la función se añade el parámetro *b,* allí se agrega como prefijo de este argumento el carácter amperstand (&).

Esta notación permite representar el *Bloque* asociado al método como un parámetro del mismo método, donde la invocación al *Bloque* se realizaría en este caso por medio de la palabra clave *call.* Note que lo único que varía es la definición del método pero la invocación a este permanece igual. Del mismo modo, se está en la libertad de hacer o no el llamado del *Bloque* dentro del método.

Es importante resalta que una de las características de los Bloques es su anonimato. Ellos no son pasados al método en una forma tradicional, no tienen nombre y son invocados mediante una palabra clave en vez de un método [22], lo cual permite abordar la solución de problemas desde una óptica distinta.

Esto proporciona una gran ventaja referente a la expresividad del LEDI, puesto que se pueden embeber una serie de instrucciones propias del LED dentro de un *Bloque* y estas ser evaluadas y ejecutadas, por el método al que está asociado, donde dentro del mismo pueden existir técnicas de *Metaprogramación* (las cuales se verán más adelante); lo cual es precisamente lo que se ha realizado en la implementación del LEDI en esta investigación.

5.1.2 Procs y Lambdas

Ahora bien, en esta instancia ya se tiene los conceptos necesarios para adentrarse en los tópicos de Procs y lambdas, donde básicamente la composición y el comportamiento que estos presentan se basa en la dinámica que manifiestan los *Bloques*. Dicho esto, se puede decir que es posible [22] *crear un objeto que represente un Bloque. Dependiendo en cómo este objeto es creado, puede ser llamado proc o lambda.*

No obstante, aunque no se haya abordado un tema que haga referencia a la creación y construcción de un Proc, este ya ha sido tocado, aunque no de una forma tan explícita como se hará a continuación.

Un objeto de tipo *Proc* puede ser creado asociando un *Bloque* con un método y utilizando el carácter ampersand (&) como prefijo de un argumento del método, tal como se hizo en (código 5.4). Sin embargo, existe otra manera de crear un objeto *Proc,* la cual es empleando el método *new* para la construcción de instancias de clase, como se muestra a continuación.

$$p \;=\; Proc.new \;\; \{\; |x, y| \;\; x + y \;\}$$ *(Código 5.5)*

Fuente: [22]

En el anterior fragmento de código, *p* es una instancia de la clase *Proc* donde este objeto representa el *Bloque* que está asociado a él, de esta forma se puede crear un método que reciba un argumento que tenga como prefijo un ampersand y sea un *Bloque*.

```
def use_proc_object(&p)
    p.call
end
```

(Código 5.7)

Fuente: [22]

Así mismo, existe *otra técnica para crear objetos Proc* y *es empleando el método lambda,* como se muestra a continuación.

```
is_positive = lambda { |x| x > 0 }
```

(Código 5.8)

Fuente: [22].

Del mismo modo, aunque la diferencia que existe en *Proc* y *lambdas* no parece sustancial a simple vista, esta se puede evidenciar en el comportamiento que manifiestan cuando son invocados, donde *llamar a un proc es similar a ceder el control a un bloque, mientras que llamar a un lambda se asemeja a la invocación de un método [22].*

Para observar mejor la diferencia de comportamiento entre un *Proc* y un *lambda*, se utilizarán algunos ejemplos que permitirán analizar la dinámica que ocurre cuando se emplea la instrucción *return* dentro de un bloque.

```
def test
    puts "entering method"
    p = Proc.new  { puts "entering proc";  return }
    p.call  #Invoking the proc makes method return
    puts  "exiting method"  #This line is never executed
end

test #invoking test method
```

(Código 5.9)

Fuente: [22].

En (c*ódigo* 5.9) se muestra la creación de un objeto Proc dentro del método llamado *test,* es importante conocer que en Ruby cuando existe un método que contiene o encierra a un *Bloque* (ya sea un Proc o un lambda) este es llamado método envolvente (*enclosing method)* o encierre léxico(*lexically enclosing),* de forma que cuando un método es invocado, el flujo lógico de la aplicación entra en el *alcance* o *contexto* del método referenciado y posteriormente al *Bloque* cuando este es llamado a través de la sentencia *p.call.*

Sin embargo, cuando el *Bloque* posee la instrucción *return,* esto no solo permite que se retorne un valor desde el propio contexto del Bloque, sino que también causa que se [22] *retorne desde el método envolvente.* En otras palabras, cuando un *Bloque* que se encuentra dentro de un *método envolvente* y posee la instrucción *return*, el flujo lógico que lleva la aplicación sale del contexto de dicho *método*, por lo tanto la instrucción *puts "existing method"* (como se muestra en (código 5.9)) nunca es ejecutada.

```
def procBuilder(message)  #Create and return a proc
    p = Proc.new  { puts message;  return }  #return returns from procBuilder
    #but procBuilder has already returned here!
end

def test
    puts "entering method"
    p = procBuilder("entering proc")
    p.call  #Prints "entering proc" and raises LocalJumpError
    puts  "exiting method"  #This line is never executed
end

test #invoking test method
```

(Código 5.10)

Fuente: [22].

No obstante, ya que es posible pasar un *Bloque* como un parámetro de función se debe tener precaución al emplear la instrucción *return* dentro del mismo y ser consciente del comportamiento que tiene este dentro un *método envolvente*.

Como se observa en (código 5.10), existe un *método envolvente* llamado *procBuilder*, el cual a su vez es invocado por el método *test;* recuerde que al emplear un *Bloque* que es de clase *Proc*, el flujo lógico es cedido al *Bloque* y al ejecutar la instrucción *return* este control sería devuelvo no solo desde el mismo *Proc*, sino desde el propio *método envolvente*, por lo tanto al invocarse el método *procBuilder* el *Bloque* residente en el ya ha retornado el control del flujo desde el contexto de este método, por lo tanto al realizar el llamado al objeto *Proc* desde el método *test* (mediante la instrucción *p.call*) se genera el error *LocalJumError,* el cual significa que el control de flujo del algoritmo ya ha sido retornado y la instrucción *p.call* estaría llamando un flujo que ya no se encuentra en ese punto.

Así mismo, esto significa que un *Bloque* al convertirse en un objeto puede ser utilizado *"fuera del contexto"* del método que lo emplea, por tanto se puede decir que un *Bloque* es una *función anónima* (como se mencionará más adelante) ya que su comportamiento no está atado al contexto del método sobre el cual se invoca.

Por otro lado, si lo que se desea es un comportamiento semejante al presentado cuando se emplean métodos pero al mismo tiempo se desea seguir utilizando *Bloques*, el enfoque a seguir es el propuesto por un *lambda.* Así pues, para [22] *crear un objeto Proc se utiliza el método lambda* (como se muestra en el código 6.8), por tanto el objeto retornado tomaría este mismo nombre.

```
def test
    puts "entering method"
    p = lambda { puts "entering lambda"; return }
    p.call #Invoking the lambda does not make the method return
    puts "exiting method" #This line *is* executed now
end
```

(Código 5.11)

Fuente: [22]

Como se observa en (código 5.11), se ha creando un objeto *lambda* nombrado como *p,* donde el *Bloque* que conforma este objeto también posee la instrucción *return.* Sin embargo, el fragmento de código que sigue a la invocación del *lambda* (un Bloque *lambda* se puede llamar de igual forma que uno de tipo *Proc*, por medio de la instrucción *p.call)* es ejecutado, siguiendo de esta forma el flujo de la lógica presentada por el método (en este caso llamado *test*); caso distinto al mostrado en (código 5.9).

De igual forma, en (código 5.12) se muestra la invocación de un *método envolvente* (llamado *lambdaBuilder*) dentro del método *test*; como puede observar este ejemplo es similar al expuesto en (código 5.10), sin embargo el empleo del *lambda* cambia radicalmente el comportamiento del código.

Observe que el método *lambdaBuilder* retorna un objeto *lambda (nombrado por medio de la variable l), que a su vez hace el llamado al Bloque lambda por medio de la instrucción l.call (como ya se ha visto anteriormente). No obstante, una vez el Bloque es llamado el flujo lógico continúa y la instrucción posterior (puts "exiting method") es ejecutada.*

```
def lambdaBuilder(message)  #Create and return a lambda
    lambda  { puts message;  return }  #return returns from the lambda
end

def test
    puts "entering method"
    l = lambdaBuilder("entering lambda")
    l.call  #Prints "entering lambda"
    puts  "exiting method"  #This line is executed
end

test #invoking test method
```

(Código 5.12)

Fuente: [22]

Del mismo modo, también se puede observar que a diferencia del ejemplo mostrado en (código 5.10), no se llega a incurrir en el error *LocalJumError* puesto que como se dijo anteriormente los [22] *lambdas trabajan semejando más a los métodos que a los bloques. Una instrucción return en un lambda, retorna desde el contexto del propio lambda, y no desde el método que rodea el lugar de creación del lambda.*

5.2 REFLEXIÓN Y METAPROGRAMACIÓN

Durante el proceso de investigación de este proyecto, Ruby ha demostrado ser un lenguaje versátil y maleable, el cual brinda un abanico de posibilidades a la hora de crear un LEDI. Del mismo modo, esta versatilidad es apoyada por la propia sintaxis del lenguaje y sobre todo por la empleo de *Bloques* a través de los objetos *Procs* y *lambdas*, los cuales al permitir representar un fragmento de código a través de un objeto, posibilitan junto con el empleo de la *Metaprogramación* agregar dinamismo y capacidad de adaptación (evolución del Modelo de Dominio) al LEDI.

5.2.1 Tiempo de Ejecución y Tiempo de Compilación, conceptos fundamentales de la Metaprogramación

Antes de adentrarse en el concepto fundamental de la *Metaprogramación*, es importante la distinción entre *tiempo de compilación y tiempo de ejecución*, de esta forma podrá comprender en qué contexto tiene su accionar esta característica de Ruby. Como se menciona en [35], la compilación o *tiempo de compilación* son t*odas aquellas etapas por las que se debe pasar un programa escrito en un lenguaje de programación, hasta poder ser ejecutable* (Fig. 4). Además, [36] *al conocer en el momento de la compilación donde va a residir el proceso en memoria se puede generar código absoluto (con direcciones absolutas).* Así mismo, al hablar de *tiempo de ejecución* se está haciendo referencia precisamente a la ejecución del programa.

Ahora, tanto en *tiempo de compilación* como en *tiempo de ejecución* puede ocurrir la reasignación de memoria. En *tiempo de compilación ocurre que* [36] *si en algún momento se desea cambiar la ubicación del proceso en memoria, se debe recompilar el código* puesto que en primera instancia se realizo la asignación de memoria empleando direcciones absolutas.

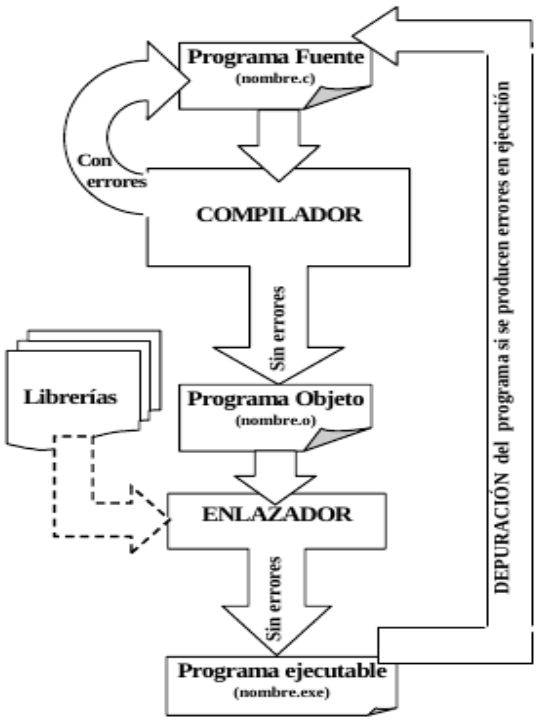


Fig. 4. Transformación de un programa fuente a un programa ejecutable [35].

Por otro lado, en *tiempo de ejecución* el sistema operativo reasigna los datos y las instrucciones a direcciones de memoria. Dicho [36] *proceso puede variar su ubicación en memoria durante la ejecución, por tanto es necesario retardar su asignación a direcciones absolutas hasta el momento de la ejecución.* Esto significa que si un lenguaje de programación posee esta característica dentro de su comportamiento, permitiría tener código fuente que evaluará fragmentos de su propio código precisamente por el hecho de que el proceso de asignación de memoria ha sido pospuesto hasta el momento de la ejecución. De esta forma, se tiene una aproximación al concepto de *Metaprogramación* que se verá más adelante.

No obstante, es conveniente una hacer un paralelo con un lenguaje conocido (C++ para este caso) para tener una mejor idea del concepto planteado. C++ es ubicado dentro de lo que han sido llamados lenguajes compilados, los cuales como su nombre lo indica, han pasado por las etapas de compilación y asignación de direcciones absolutas, esto significa que una vez el compilador a terminado su trabajo, miembros tales como variables y métodos pierden su carácter concreto; estos solo ya serían posiciones en memoria.
Es decir, al momento de preguntarle a una clase por sus métodos de instancia, esta no sabría cómo responder ya que estos habrían sido convertidos a direcciones absolutas y por lo tanto la clase se ha desvanecido. Por otro lado, Ruby no posee tiempo de compilación y la mayoría de construcciones que realiza un programa creado en este lenguaje estarán disponibles en tiempo de ejecución, lo cual abre paso a la definición del concepto de *Metaprogramación*.

5.2.2 Metaprogramación

Ahora bien, haciendo ya una introducción formal a este tema la *Metaprogramación* parece ser un concepto complejo y difuso al inicio, especialmente si ve viene de la vertiente de los lenguajes originados desde C. Sin embargo, Ruby al ser un lenguaje que reside su funcionamiento en la etapa de *tiempo de ejecución* permite crear programas que sean dinámicos, ya que como se mencionó anteriormente, la mayoría de las *construcciones* (es decir, definiciones de clases, métodos, objetos, etc) que este realiza se encuentran en dicha etapa. Cuando se hace referencia al término *construcciones*, una forma para comprenderlo mejor es haciendo una analogía con el método llamado *constructor*.

Como es sabido, un *constructor* es una función especial perteneciente a una clase que es llamado de forma automática al crear un objeto y es por medio del cual se reserva un espacio en memoria para la construcción del objeto (tenga en cuenta que se está haciendo referencia en este ejemplo a un lenguaje compilado).

Anteriormente se mencionó que una vez el código fuente es compilado, todos aquellos objetos y los componentes que los caracterizan son convertidos a *direcciones absolutas* de modo que no existe posibilidad que dentro del *tiempo de ejecución* se puedan construir más clases, objetos, métodos, etc. En cambio, Ruby si puede realizar estas *construcciones* en *tiempo de ejecución.*

Dicho esto, se puede definir a la *Metaprogramación* como *la escritura de código que manipula las construcciones del lenguaje en tiempo de ejecución [34].* Adicionalmente, la *Metaprogramación* no es un comportamiento externo que sea añadido a la dinámica de Ruby en busca de una funcionalidad específica. De hecho, está estrechamente ligada al lenguaje y es empleada continuamente en la programación "regular".

Un ejemplo de esto es el método *attr_accessor el cual es empleado para definir los métodos que permitirán asignar y obtener los valores que poseerán los atributos de instancia de clase, es decir, este método reemplaza los métodos setters y getters* que son típicos en la definición de una clase en lenguaje como Java o C++.

Como se observa en (código 5.13), el método *attr_accessor* toma como parámetro los atributos de clase *x* y *y.* Posteriormente, se hace el llamado al método *instance_methods8* para mostrar todos los métodos de instancia de la clase (*Point.instance_methods(false)*), de este modo el programa muestra un arreglo de la forma [:x, :x=, :y, :y=]

En este arreglo cada uno de los elementos representan el método g*et* y s*et* de las variables de instancia *x* y *y*. Así pues, el símbolo *:x* corresponde al método *get* de *x* y el símbolo *:x=* corresponde al método *set* de *x*; lo mismo ocurriría con la variable *y.*

```
class Point
    attr_accessor :x, :y   # Define accessor methods for our variables
end
```

Cógigo 5.13)

```
puts "#{Point.instance_methods(false)}"   #Prints[:x, :x=, :y, :y=]
```

8 Retorna una matriz que contiene los nombres de los métodos de instancia públicos y protegidos en el receptor. Si el parámetro opcional es falso (false), no se incluyen los métodos de los heredados. http://ruby-doc.org/core-2.2.0/Module.html#method-i-instance_methods

Fuente: Alejandro Rodas Vásquez

Por otro lado, si no se deseara emplear el método *attr_accessor* un camino alternativo sería definir cada uno de los métodos *set* y *get* para cada uno de los atributos de la clase, como se muestra en (código 5.14).

```
class MutablePoint
    def initialize(x, y); @x, @y = x, y; end

    def x; @x; end   #The getter method for @x
    def y; @y; end   #The getter method for @y

    def x = (value)   #The setter method for @x
       @x = value
    end

    def y = (value)   #The setter method for @y
       @y = value
    end
end
```

(Código 5.14)

Fuente: [22].

Aunque los programas mostrados en (código 5.13) y (código 5.14) son equivales, el primero es más conciso y funcional. Sin embargo, puede surgir la incógnita ¿cómo hace Ruby para generar todos los métodos *setters* y *getters* de (código 5.14)? Pues bien, como se mencionó anteriormente, Ruby tiene una estrecha relación con la *Metaprogramación* y hace uso de la misma en muchas de sus instrucciones básicas, tal es el caso del método *attr_accessor* el cual emplea la *Metaprogramación* para crear dinámicamente los métodos *set* y *get* mostrados en la (código 5.14).

Esto demuestra que si se alcanza tener dominio sobre la *Metaprogramación* o mejor aún, sobre alguna de las técnicas que hacen uso de ella, se lograría añadir un complemento importante a la arquitectura del LEDI permitiendo sumar versatilidad y dinamismo no solo a la sintaxis sino al comportamiento del lenguaje en construcción.

5.3 TÉCNICAS DE METAPROGRAMACIÓN EMPLEADAS EN EL PROYECTO

Como se dijo en el anterior apartado, conocer algunas de las técnicas que hacen uso de la *Metaprogramación* es un componente importante a la hora de crear lenguajes que tengan como *lenguaje anfitrión* a Ruby (tal es el caso de Ruby on Rails). En esta sección se describe la técnica llamada *Definición dinámica de métodos*, haciendo una descripción detallada de su funcionamiento y cómo este permite crear un componente esencial dentro de la arquitectura que es objeto de la presente investigación.

Como ya se ha definido la *Metaprogramación es la escritura de programas (o frameworks) que ayudan a escribir programas*, lo que significa que Ruby al ser un *lenguaje dinámico* permite dentro de su comportamiento insertar métodos dentro de una clase en tiempo de ejecución, de igual forma crear *alias*[9] para los métodos existentes e incluso definir métodos para un objeto en particular. No obstante, para lograr esto todo aquel programa que se esté codificando y vaya a hacer uso de las propiedades provistas a través de la *Metaprogramación*, debe *poder examinar su estado y estructura* de modo que sea capaz de modificarlos o alterarlos. Este comportamiento como tal, es llamado *Reflexión (Introspección)*. Un ejemplo de la aplicación de este concepto se puede observar en (código 5.13) cuando se emplea el método *instance_method*, el cual evalúa el estado de la clase y consultó cuales son los métodos de instancia de la misma.

Crear un programa que utilice la *Metaprogramación* (o sea, crear un programa que pueda escribir programas) implícitamente debe hacer uso de aquellas funcionalidades que provee Ruby a través de la *Reflexión,* las cuales permitirán evaluar y modificar el propio programa según el objetivo requerido.

Así pues, en este apartado se mostrarán las técnicas o métodos utilizados en la presente investigación a través de los cuales se implementa la *Metaprogramación* para alcanzar un LEDI adaptable y expresivo.

Una de las principales dificultades que se tuvo para alcanzar estos atributos, fue la necesidad de encontrar una técnica que permitiese la *creación dinámica de métodos*, esto se debe a que uno de las características que posee el LEDI que se pretende construir, consiste en *permitir crear cualquier número de restricciones*, donde cada restricción en realidad es un método que acepta como parámetro la inecuación que expresa la propia restricción; como se puede observar en (código 5.15).

9 No es raro que los métodos en Ruby tengan más de un nombre. El lenguaje tiene la palabra clave *alias* que sirve para definir un nuevo nombre para un método existente. [22]

$$
\begin{array}{l}
max('2x1 + 0.34x2').subjectTo\ \{ \\
\quad restrictionA \quad '0.03x1 > 0.98' \\
\quad restrictionB \quad '10.6x1 <= 0.002' \\
\quad restrictionC \quad '2x1 >= 1.57' \\
\quad \}
\end{array}
$$

(Código 5.15)

Fuente: Alejandro Rodas Vásquez

Observe en el (código 5.15) la definición de las restricciones sujetas a la función de optimización *(restrictionA, restrictionB, restrictionC)*. Cada una de estas restricciones como se mencionó anteriormente, son métodos que reciben como argumento una cadena de caracteres que expresan una inecuación. De esta forma, haciendo uso de una de las características de Ruby, donde el empleo de paréntesis en el llamado a una función no es obligatorio, se logra alcanzar un nivel de expresividad en la instrucción.

Por otro lado, restringir al usuario no solo en el número sino en el nombre que debe tomar la restricción, es una limitación que puede ser sorteada a través de la *creación dinámica de métodos*.

Definición dinámica de métodos

Usualmente se está acostumbrado a tener a los métodos como elementos identificables dentro de una clase de modo que al hacer la invocación de alguno de ellos ya se tiene una idea a quien pertenecen. Sin embargo, es posible a través de la implementación de la *Metaprogramación* lograr crear métodos dinámicamente, de modo que sea posible invocarlos sin necesidad de estar definidos (como tradicionalmente se conoce) en una clase. Para lograrlo se hará uso del método llamado *define_method* que es propio de Ruby.

```
class MyClass
    def define_method :my_method do |my_arg|
       my_arg * 3
    end
end
```

(Código 5.16)

```
obj = MyClass.new
obj.my_method(2) # => 6
```

Fuente: Alejandro Rodas Vásquez

Como se observa en (código 5.16), *define_method* recibe al símbolo *:my_method*, este primer parámetro representa el nombre del método que dinámicamente se esta definiciendo, por lo tanto el método que se está creando en la figura tomará el nombre de *my_method*. Posteriormente, se puede notar que existe un bloque asociado, donde este bloque representa el cuerpo del método que se define.

De esta forma, se ha construido un método que no ha sido definido explícitamente en la clase pero que sin embargo puede ser creado e invocado en tiempo de ejecución, permitiendo ser reconocido como un método de instancia de la clase *MyClass*, como muestra en (código 5.16)

Por otro lado, un método perteneciente a Ruby y que de igual forma es importante dentro del funcionamiento del LEDI del presente proyecto, ha sido el método *send.* Este ofrece una forma distinta de invocación a los métodos definidos dinámicamente (como se verá más adelante) o explícitamente. Para comprender mejor esto, observe a (código 5.17) donde se tiene la clase *MyClass* que define de manera explícita el método llamado *my_method.*

```
class MyClass
    def my_method(my_arg)
       my_arg * 2
    end
end

obj = MyClass.new
obj.my_method(3)  # => 6
```

(Código 5.17)

Fuente: [34]

Hasta este momento no se muestra nada distinto; más sin embargo, a continuación observe cómo se realiza la invocación de *my_method* por medio del método *send* obteniendo el mismo resultado.

```
obj.send(: my_method , 3)  # => 6
```

(Código 5.18)

Fuente: [34]

Como puede notar, el primer argumento de método *send*, es el mensaje que se está enviado al objeto, *en este caso el nombre del método.* Los argumentos restantes (ya sea un *Bloque*, si este existe) son pasados al método. Esta forma de invocación al método permite implementar un estilo de programación donde el llamado a una función no se hace de forma explícita, lo cual restringe la adaptabilidad del código (precisamente lo que permite la implementación de la *Metaprogramación* y la *Reflexión*) a distintas situaciones, permitiendo así [34] *esperar hasta el último momento para decidir qué método llamar, mientras el código es ejecutado. Esta técnica es llamada Envío Dinámico (Dynamic Dispatch), la cual será de gran utilidad en la construcción del LEDI.*

Para dar una mejor perspectiva de la forma en cómo se puede emplear el método *send* se presenta el siguiente ejemplo. En (código 5.19) se tiene la clase *Computer* donde cada uno de sus método define los componentes que un computador puede tener, esto se realiza por medio de la invocación al método llamado *component* que toma como argumento la parte o sección del computador que está definiendo la función.

```
class Computer
    def initialize(computer_id, data_source)
        @id = computer_id
        @data_source = data_source
    end

    def mouse
        component :mouse
    end

    def cpu
        component :cpu
    end

    def keyboard
        component :keyboard
    end

    def component(name)
        info = @data_source.send "get_#{name}_info", @id
        price = @data_source.send "get_#{name}_price", @id
        result = "#{name.to_s.capitalize}: #{info} ($#{price})"
        return "* #{result}" if price >= 100
        result
    end

end
```

(Código 5.19)

Fuente: [34]

Ahora, observe la implementación del método *component.* Básicamente este método permite invocar a cualquiera de los métodos *get* que permiten obtener la información (*get_#{name}_info*) y el precio (*get_#{name}_price)* definidos en la clase *DS,* la cual es referenciada por medio del objeto instanciado en la variable *@data_source.* El valor de la variable de instancia *@data_source* es asignado por el segundo parámetro (*DS.new*) del constructor de la clase Computer (código 5.20).

```
my_computer = Computer.new(42, DS.new)
my_computer.cpu   # => * Cpu: 2.16 Ghz ($220)
```

(Código 5.20)

Fuente: [34]

Del mismo modo, si no se hiciese uso del método *send* para lograr la implementación de *component,* se tendría que colocar una serie de condicionales que abarcasen todos componentes *(mouse, cpu y keyboard)* que posea la clase *Computer* y posteriormente hacer el llamado de forma explícita de los métodos *get* asociados al componente a través del objeto *@data_source*.

De esta forma, implementar la *técnica de Envío Dinámico* por medio del método *send* no solo reduce notablemente el número de líneas de código sino que añade flexibilidad y adaptabilidad al programa, ya que se evita caer en la estructura rígida que implica diseñar un código para un comportamiento explícito.

Ahora bien, una vez visto el empleo de la función *define_method* en la creación de métodos definidos de forma implícita y la implementación de la *técnica de Envío Dinámico* a través del método *send*, es apropiado mostrar por medio de un ejemplo, cómo se pueden integrar estos dos recursos de modo que se pueda tener dentro de la estructura del programa un fragmento de código que permita implementar la *Generación Dinámica de Métodos. La cual es una funcionalidad importante dentro de la arquitectura que sustenta al LEDI del presente proyecto.*

Observe en (código 5.21) las últimas tres líneas de código las cuales hacen el llamado al método *define_component,* el cual es definido como un método de clase por medio de la palabra reservada *self.* Note que *define_method* es invocado dentro de la definición de la clase *Computer* por lo tanto todos los métodos creados por medio de este serán definidos como *métodos de instancia*.

Por otro lado, dentro de *define_method* se tiene implementada la misma funcionalidad que se mostró en (código 5.19) de modo que se obtendrá el mismo comportamiento ya explicado.

```
class Computer
    def initialize(computer_id, data_source)
        @id = computer_id
        @data_source = data_source
    end

    def self.define_component(name)
        define_method {
            info = @data_source.send "get_#{name}_info", @id
            price = @data_source.send "get_#{name}_price", @id
            result = "#{name.to_s.capitalize}: #{info} ($#{price})"
            return "* #{result}" if price >= 100
            result
        }
    end

    define_component :mouse
    define_component :cpu
    define_component :keyboard
end
```

(Código 5.21)

Fuente: [34]

Cabe recordar que como el método *define_component* se encuentra dentro del *Ámbito Interno* de la clase, una vez que se crea una instancia de la misma, este método será invocado y creará dinámicamente los métodos llamados *mouse*, *cpu* y *keyboard*, por cada llamado.

Por consiguiente, la clase *Computer* podrá invocar estos tres métodos (código 5.22) sin necesidad de haberlos creado explícitamente.

```
obj = Computer.new(42, DS.new)
obj.cpu
obj.mouse
obj.keyboard
```

(Código 5.22)

Fuente: [34]

***Búsqueda de Método* (*Method Lookup*)**

Ahora, hasta el momento se ha podido observar en varias ocasiones cómo se define una clase y la invocación de los métodos pertenecientes a ella, ya sean creados explícitamente o dinámicamente. Sin embargo, para poder comprender los componentes que son parte de la arquitectura base del LEDI construido en esta investigación es necesario entender qué sucede cuando Ruby busca los métodos que son invocados. De esta forma, se llega al concepto llamado *Búsqueda de Método o Resolución de Nombres de Método (Method Lookup or Method Name Resolution*).

Cuando [22] Ruby evalúa una expresión que llama a un método se debe primero descubrir qué método es el invocado. El proceso para realizar esto es llamado Búsqueda de Método o Resolución de Nombres de Método.

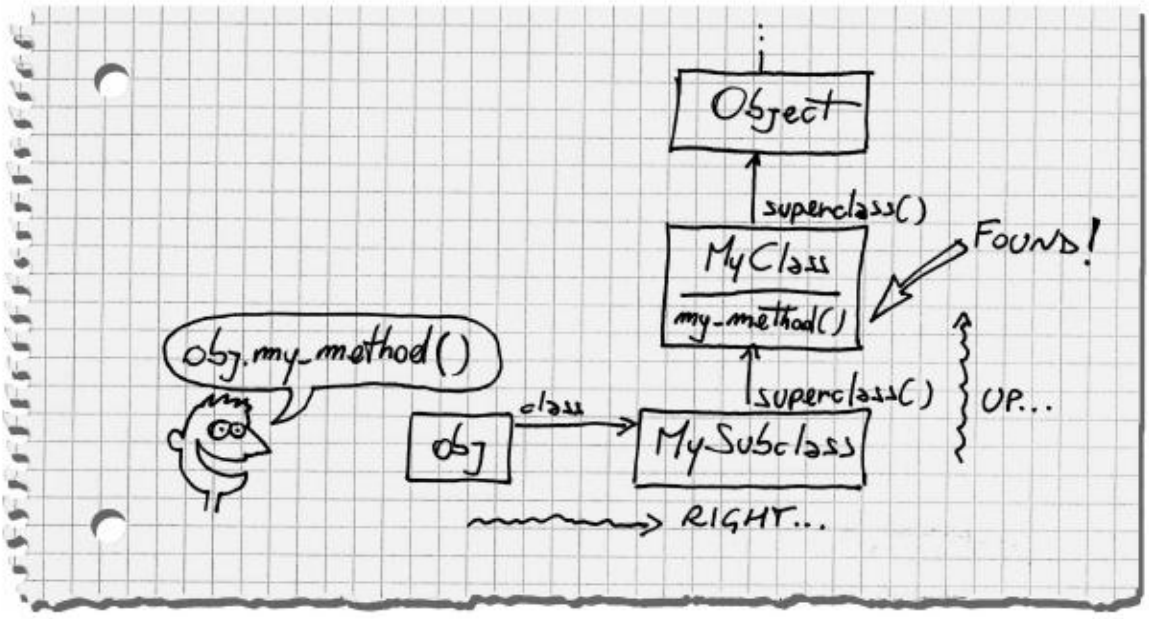


Fig. 5. Búsqueda de un método en la Cadena de antepasado [34].

Básicamente para entender el funcionamiento de este proceso se debe tener claro los conceptos de receptor (*receiver*) o destinatario y cadena de antepasados (*ancestors chain*).

El receptor o destinatario es *simplemente el objeto que se llama para invocar a un método.* Por ejemplo, asuma que se tiene la siguiente instrucción *my_array.rotate()*, en esta sentencia el objeto *my_array* es denominado como *receptor.*

Del mismo modo, tener una mejor concepción de lo que ha sido llamado *cadena de antepasados* sitúese en cualquier clase de Ruby [34], ahora imagine que estando en la presente clase se pasa a la superclase de la misma, luego se dirige hacia la superclase de la actual superclase y así sucesivamente hasta llegar a la clase *Object* (la cual es la superclase por defecto) y finalmente a la clase *BasicObject* (la cual es la raíz de la jerarquía de clases de Ruby). De esta forma, el camino que se transita desde la clase del objeto hasta la superclase que aloja el método invocado se denomina *cadena de antepasados de la clase*.

Ahora, sabiendo el concepto de *receptor* y comprendido qué es lo que implica cuando se hace referencia al término de *cadena de antepasados de la clase*, es posible tener una idea más detallada del proceso que inicia Ruby al momento de realizar la *Búsqueda de Método* invocado por un objeto.

En primer lugar (Fig. 5), Ruby se dirige a la clase a la cual pertenece el *receptor* y desde este punto empieza a escalar por todo la *cadena de antepasados* de esta clase hasta lograr encontrar el método invocado.

A continuación, se muestran las etapas que sigue Ruby para lograr una *Resolución de Nombres de Métodos* (según [22]). Téngase en cuenta que para esta explicación se empleará la expresión *o.m* para la invocación del método; donde *o* es el objeto instanciado (receptor) y *m* es el método invocado.

1. Primero, se chequea la clase denominada *eigenclass* del objeto *o* en búsqueda del método *singleton* llamado *m.*
2. Si no se encuentra el método *m* en la clase *eigenclass*, Ruby busca en clase a la que pertenece el objeto *o* con el objetivo de encontrar el método de instancia *m.*
3. Si el método *m* no es encontrado en esta clase, Ruby busca el método de instancia en cualquier *módulo*[10] que haya sido incluido por la clase a la que

10 Los *Módulos* se definen de forma similar a un clase (en Ruby) pero la palabra reservada *module* es empleada en lugar de *class.* A diferencia de una clase, un *módulo* no puede ser instanciado y tampoco ser una subclase [22].

pertenece el objeto *o.* Si esta clase incluye más de un *módulo,* la búsqueda se realiza en cada uno de los módulos incluidos en el orden en que ellos hayan sido agregados a la clase.

4. Si el método de instancia *m* no se encuentra ni en la clase a la que pertenece el objeto *o* ni en ninguno de los *módulos* incluidos, la búsqueda se dirige hacia la jerarquía de herencia de la superclase. Los pasos 2 y 3 son repetidos para cada una de las clases pertenecientes a esta jerarquía hasta llegar a la clase ancestro de todos aquellos módulos que esta pueda incluir.
5. Si no se encuentra ningún método llamado *m* luego de realizada la búsqueda, se invoca posteriormente al método nombrado como *method_missing.* Con el afán de encontrar una apropiada definición de este método, el algoritmo de *resolución de nombres* comienza de nuevo desde el paso 1.

De esta manera, conocer de forma detallada el proceso que realiza Ruby al momento de llamar un método no solo es importante para entender el comportamiento del lenguaje sino también como una herramienta fundamental para la manipulación de este aprovechando las bondades que ofrece como *lenguaje anfitrión.*

En el siguiente capítulo se abordan los tópicos *Singleton Method* e *Eingenclass*, los cuales son características propias de Ruby que intervienen directamente en la *Búsqueda de Método* y por consiguiente en la *cadena de antepasados de la clase.*

5.4 SINGLETON METHOD Y *EINGENCLASS*

Como se ha señalado en el apartado anterior, Ruby inicia el proceso de búsqueda del método invocado recorriendo cada una de las clases y módulos que son incluidos en ellas. Sin embargo, en el primer paso de este proceso se hace referencia a dos elementos importantes dentro del funcionamiento básico del lenguaje los cuales son la clase denominada *Eingenclass* y del mismo modo el método nombrado como *singleton.*

De tal forma, en este apartado se explicaran estos conceptos ya que aparte de estar involucrados en el proceso de *Búsqueda de Métodos* (el cual fue necesario comprender para desarrollar funcionalidades básicas para el LEDI), estos fueron utilizados en el diseño de un componente perteneciente a la arquitectura que sustenta al LEDI en construcción. Dicho esto, es conveniente definir qué puede ser llamado o denominado como *Singleton Method.*

Es ya conocido que toda instancia de una clase posee y puede invocar todos los métodos que en esta se definen. Sin embargo, imagine que usted necesita definir una serie de objetos que son del mismo tipo de clase, pero cada uno de estos objetos poseerá métodos que son distintos a los definidos en la clase a la cual pertenece y al mismo tiempo estos métodos pueden no ser los mismos entre los objetos.

Dicha funcionalidad puede parecer extraña y en algunos casos no tener sentido, especialmente para las personas que están acostumbradas a interactuar con lenguajes que tiene como base C. No obstante, si se piensa detenidamente en ello, esto permite crear objetos que aunque tengan una clase que les proporcione su caracterización e identidad, al mismo tiempo permite crear objetos que se manifestarían como entidades autónomas donde su comportamiento podría ser modificado mediante la agregación de métodos. Para comprender mejor lo dicho se analizará el siguiente fragmento de código.

```
1   class Point
2     def my_method; "This is an instance method" end
3   end
4
5   first_obj = Point.new
6   second_obj = Point.new
7                                                          (Código 5.23)
8   def first_obj.sample_method
9     "Sample Method for class #{self.class.name}"
10  end
11
12  puts  first_obj.my_method   # => "This is an instance method"
13  puts  first_obj.sample_method   # => "Sample Method for class Point"
14  puts  second_obj.sample_method   # => "singleton_method.rb : 14 : in ‘ < main >′:
                                        undefined method ‘sample_method′ for
                                        # < Point : 0x00000000c5ba20 >  (NoMethodError)"
```

Fuente: Alejandro Rodas Vásquez

Como se observa en (código 5.23), se tiene la definición de la clase *Point* la cual posee un método nombrado como *my_method*. Del mismo modo, en las líneas 5 y 6 se crean las instancias de clase *first_obj* y *second_obj* las cuales pueden invocar el método *my_method, como se muestra en la línea 12. Sin embargo, en la línea 8 se puede encontrar la definición del método llamado como sample_method, el cual no pertenece a la clase, donde al observar detenidamente dentro de la definición de este, se puede notar que se hace referencia al objeto first_obj, esto significa que solamente este objeto puede hacer uso de este método. Esto se puede constatar en la línea 13, donde se hace un llamado a sample_method arrojando como resultado el mensaje "Sample method for class Point". Caso contrario al que se representa en la línea 14, en donde el objeto second_obj al tratar de realizar la invocación al mismo método obtiene un mensaje de error ("singleton_method.rb:14:in `<main>': undefined method `sample_method' for #<Point:0x00000000c5ba20> (NoMethodError)") que advierte que este no esta definido para tal objeto.*

Por lo tanto, un método como *sample_method* el cual [34] *está especificado para un solo objeto, es llamado método Singleton.*

Ahora, ya que se conoce que un método *Singleton* está asociado a un objeto en particular, surge la pregunta ¿en donde residen o se pueden encontrar los métodos *Singleton*, ya que un método de este tipo no puede existir dentro de un objeto ni tampoco dentro de una clase? como se mostró en (código 5.23). Esta pregunta permite dar paso al siguiente tópico denominado *Eingenclass.*

Anteriormente se mencionó que en el proceso de *Resolución de Nombres de Métodos* el primer paso consiste en chequear la llamada *Eingenclass* del objeto que está invocando al método. Sin embargo, al realizar la búsqueda de esta clase ella no se encuentra a simplemente vista, ni está definida dentro de la *cadena de antepasados* del objeto de tal forma que pareciera que fuera una clase *anónima.* Teniendo en cuenta esto, remítase de nuevo a la línea 9 en (código 5.23), allí puede encontrar la sentencia *self.class.name*, la cual se emplea para revelar la clase a la que pertenece un objeto, dando como resultado el mensaje que se muestra en la línea 13. Pues bien, al utilizar esta instrucción, Ruby no está revelando todas las clases a las que pertenece un objeto; [34] *a parte de la clase que se puede observar, un objeto puede tener su propia clase especial, una clase oculta. Esta es la denominada clase Eingenclass del objeto.*

Acto seguido, observe el fragmento de código presentado en (código 5.24). Cuando se necesite obtener la *Eingenclass* de un objeto se utilizará la palabra reservada *self* para referirse a la *Eingenclass de dicho objeto, donde el objeto como tal será llamado con la letra o* (obviamente este nombre puede cambiar, sino que ha sido llamado de esta manera para efectos del ejemplo)*.*

$$eingenclass = class << o; self; end$$ *(Código* 5*.24)*

Fuente: [22]

A continuación, usando el fragmento de código que se observa en (código 5.25) se mostrará cómo un *objeto puede tener su propia clase especial*, *es decir la clase Eingenclass.*

```
1   class C
2     def a_method
3       'C#a_method()'
4     end
5   end
6
7   class D < C; end
8
9   class Object
10    def eigenclass
11      class << self; self; end
12    end
13  end
14
15  obj = D.new
16  puts obj.eigenclass.superclass   # => D
```

(Código 5.25)

Fuente: [22].

En (código 5.25) se tiene entre la línea 1 y 5 la definición de la clase llamada como *C* la cual posee el método *a_method.* Del mismo modo, en la línea 7 se encuentra la clase nombrada como *D,* la cual hereda de *C.* De esta forma, como ya es sabido toda instancia de la clase *D* (línea 15) hereda los métodos de *C.*

Por otro lado, para comprobar la existencia de la clase *Eingenclass* residente en el objeto *obj*, se define entre las líneas 9 y 13 la clase *Object* (la cual es la superclase por defecto de Ruby) donde se crea el método eingenclass el cual tiene como función proporcionar la clase *Eingenclass* del objeto instanciado.

Ahora observe en Fig. 6, donde se tiene un diagrama de clases del código presentado en (cógido 5.25). Del mismo modo, se puede notar que en esta figura no se incluye la clase *Eingenclass* del objeto, lo cual puede parecer extraño y distinto a lo que normalmente se está acostumbrado. Para aclarar esto observe la línea 16 en (cógido 5.25), allí el objeto *obj* hace un llamado al método eingenclass perteneciente a la clase *Object* y posteriormente al método *superclass*[11] mostrando como resultado que la superclase de *obj* es la clase *D.*

11 Retorna la superclase de un objeto, o nil. http://ruby-doc.org/core-1.9.3/Class.html#method-i-superclass

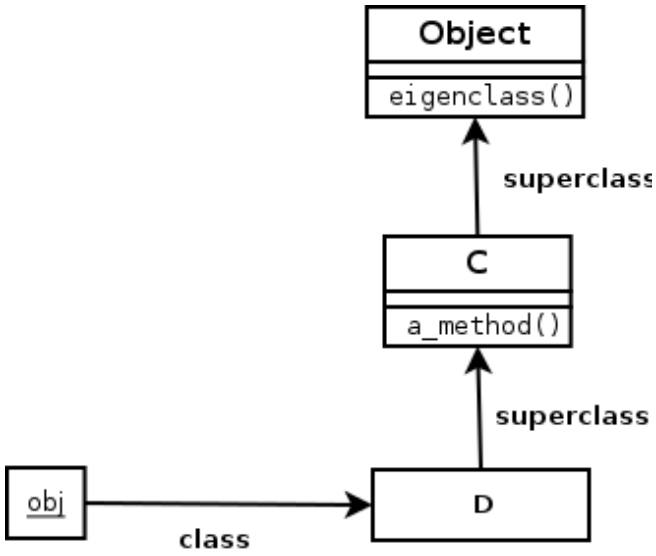


Fig. 6. Jerarquía de clases y cadena de antepasados del objeto *obj* [34].

Esto lleva a pensar que para que este resultado se dé significa que la clase *Eingenclass* del objeto en cuestión se encuentra en un nivel por debajo que la clase *D*; como se puede observar en el diagrama presentado en Fig. 7.

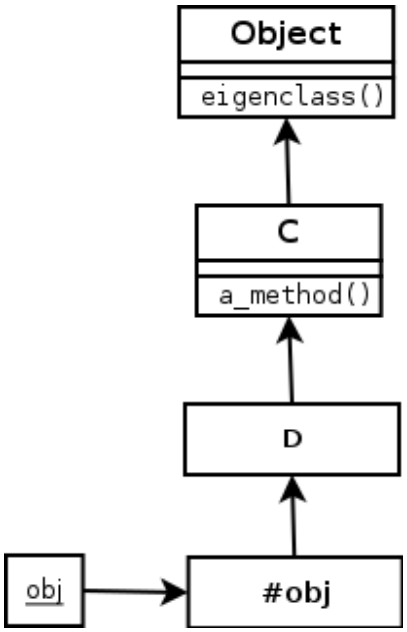


Fig. 7. Cadena de antepasados del objeto *obj.* Se antepone el carácter numeral (#) al nombre del objeto para identificar la clase *Eingenclass* del mismo [34].

Una vez comprendido tanto de lo qué es la *Eingenclass* como el concepto de *cadena de antepasados* de un objeto, ya se tiene una mejor idea de las tareas que se deben realizar en el proceso de *Resolución de Nombre de un Método* en su primera fase. Lo cual lleva a centrarse en el último paso que presenta este proceso, donde es invocado el método nombrado *method_missing*[12]*.*

Como ya ha sido mencionado, luego de completarse la búsqueda del método invocado por el objeto y no haber sido este encontrado, *method_missing* es llamado*.* Con el propósito de hallar la definición de este método por medio del algoritmo de *Resolución de Nombres de Método,* se comienza a buscar la definición de *method_missing* por todas aquellas clases y módulos que se encuentran en la *cadena de antepasados* del objeto creado.

Dicho comportamiento brinda la posibilidad de poder invocar métodos que no se encuentren definidos explícitamente en una clase ni en un módulo. De esta forma, dentro de *method_missing se podría crear un método que implementase la técnica de Generación Dinámica de Métodos, de modo que se contaría con un componente dentro de la arquitectura del LEDI que se dedicara a la construcción de métodos en tiempo de ejecución.*

Precisamente esto es lo que se ha logrado plasmar en el Anexo F del proyecto, allí se muestra el código fuente del módulo denominado *RestrictionBuilder*, el cual pertenece a la arquitectura construida para el LEDI objeto de la presente investigación. Este módulo ha sido diseñado especialmente para implementar la técnica de *Generación Dinámica de Métodos.* Como se puede observar en este Anexo, dentro de *method_missing* se hace un llamado a la clase *Eingenclass* del objeto*, la cual por medio del método class_eval* puede incluir dentro de la propia *Eingenclass* una función que haya sido creado dinámicamente empleando *define_method.* Así mismo, se utiliza el método *send* que recibe como parámetro el nombre del nuevo método, por consiguiente este se ejecutará en el mismo instante de su creación, es decir en *tiempo de ejecución.*

12 method_missing(symbol [, *args]) → result. Es invocado por Ruby cuando el objeto *obj* envía un mensaje que no puede ser manejado. s*ymbol* es el símbolo que representa el nombre del método que se invoca, y *args* son todos los argumentos que se pasan al método invocado. Por defecto, el interprete muestra un excepción cuando este método es llamado. Sin embargo, es posible sobre escribir este método para ofrecer un comportamiento más dinámico. http://ruby-doc.org/core-2.1.0/BasicObject.html#method-i-method_missing

6. PATRONES DE CONSTRUCCIÓN DE UN LENGUAJE ESPECÍFICO DE DOMINIO INTERNO (LEDI) Y SELECCIÓN DE UNO DE ELLOS PARA IMPLEMENTAR EN EL PROYECTO

Ahora, en esta instancia del documento se ha abordado el concepto de LED, se hizo hincapié en la importancia de un *Modelo de Dominio* que lo sustente, se definieron los conceptos fundamentales de la metodología C4 (las cuales se emplearán más adelante para la definición de la arquitecta del lenguaje), las técnicas de *Metaprogramación* que se emplearán en el proyecto, entre otros. Sin embargo, aún falta por tocar los tópicos referentes a las técnicas o patrones que sirven para la implementación del LEDI, el cual es el objetivo de este capítulo.

La clasificación que se hace de los Lenguajes Específicos de Dominio está relacionada con la forma en cómo estos son implementados, es decir, cada una de los enfoques propuestos, presenta características propias que se traducen en el diseño e implementación del LED. Según [6] existen dos enfoques principales de LED que son LED Interno (LEDI) y LED Externo, como se explica a continuación.

LED Interno (LEDI)

Un LED Interno es aquel que usa la infraestructura de un lenguaje de programación existente (también llamado lenguaje anfitrión) para construir la semántica de un dominio-específico [4].

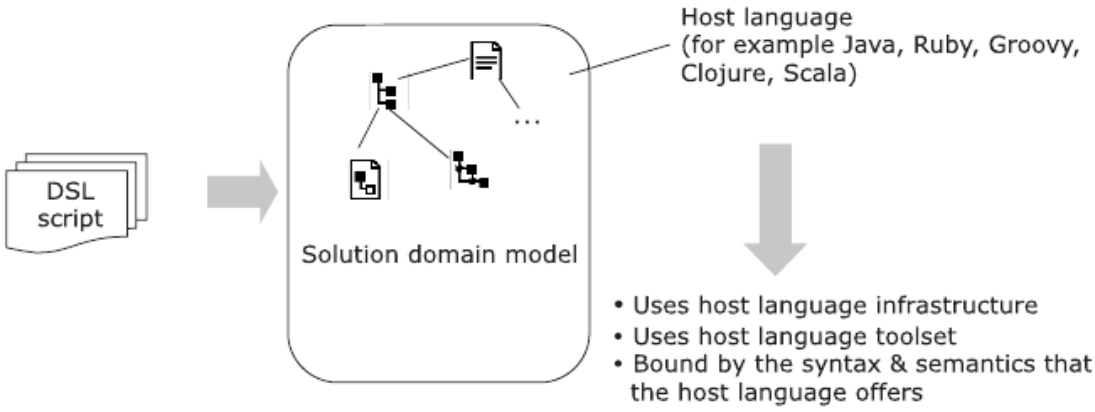


Fig. 8. Implementación de un LED Interno utilizando un lenguaje anfitrión existente y la infraestructura que el ofrece [4].

LED Externo

Un LED Externo es aquel que es desarrollado desde cero y tiene una infraestructura separada para el análisis sintáctico, técnicas de análisis, interpretación, compilación y generación de código. Desarrollar un LED Externo es similar a implementar un nuevo lenguaje desde el inicio con su propia sintaxis y semántica [4]. Como se puede observar en la figura 9, el *DSL script* tiene que pasar por dicha infraestructura para ser analizado e interpretado, para así generar una solución adaptable al Modelo de Dominio

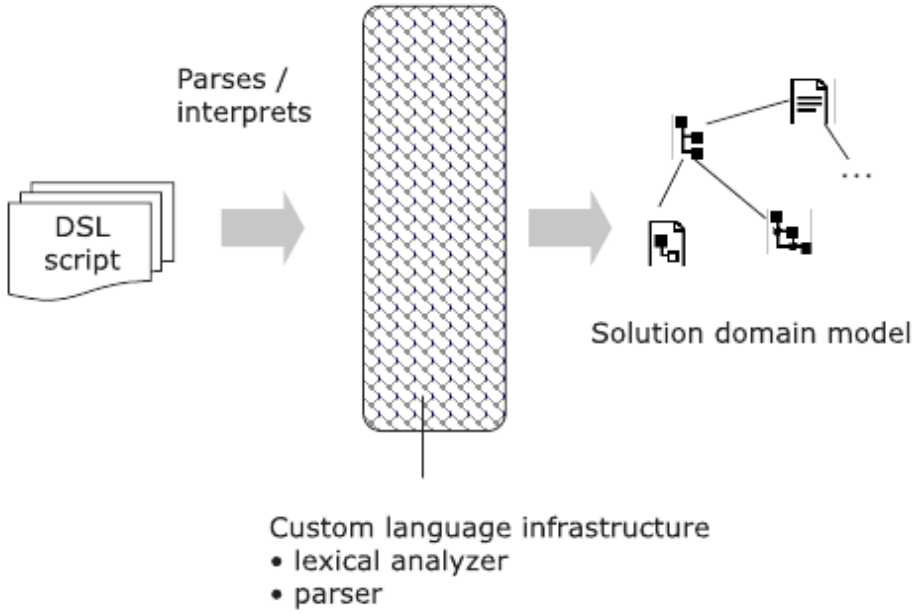


Fig. 9. Infraestructura para el procesamiento de lenguaje para un LED Externo [4].

No obstante, dado que la presente investigación *se enfoca en la creación de un LED Interno* se hará énfasis en éste y los conceptos relacionados con el mismo.

6.1 PATRONES DE CONSTRUCCIÓN DE UN LEDI

Ampliando la anterior definición de *LED Interno*, es útil especificar qué significa cuando se habla de "*usar la infraestructura de un lenguaje de programación existente"*.

Como ya es conocido, dentro de la composición que posee un lenguaje de programación no se puede olvidar mencionar su compilador, el cual posee diversas fases como lo son: *analizador léxico, analizador sintáctico, analizador semántico, etc.* Es decir, se cuenta con una infraestructura ya implementada la cual será utilizada por el LEDI.

Por tal motivo, un LEDI también puede ser llamado *Lenguaje Específico de Dominio Embebido,* refiriéndose al hecho que el LED al utilizar la infraestructura de un *lenguaje anfitrión* estará ligado a las restricciones de este, por lo tanto, cualquier expresión que se utilice debe ser un expresión legal en el lenguaje anfitrión por lo tanto es importante escoger un *lenguaje anfitrión* que sea versátil y cumpla con los criterios de expresividad requeridos; como lo ha demostrado Ruby.

Por otro lado, un atributo que distingue a un LED ya sea *Interno* o *Externo* yace en las características de los nombres que poseerán sus instrucciones. Como ya es conocido, un LED se fundamenta en el *Modelo de Dominio* de un contexto específico y busca que el usuario del lenguaje pueda expresarse en términos de este, de esta forma la semántica de los nombres escogidos es fundamental, de modo que el usuario al emplear dicha sintaxis pueda expresar el contexto del problema que trata de modelar.

Sin embargo, dichas instrucciones carecen de significado si se emplean de forma aislada puesto que su semántica se apoya en un empleo conjunto con otras instrucciones, de modo que la unión de ellas pueda expresar un contexto en particular. Para esclarecer mejor este concepto considere una situación donde se necesita modelar un problema de optimización utilizando para este caso un LEDI, como se muestra en (código 6.1).

$$
\begin{aligned}
&max(:x1 => 2, \quad :x2 => 0.34).subjectTo\ \{ \\
&\quad restrictionA \quad '0.03x1 > 0.98' \\
&\quad restrictionB \quad '10.6x1 <= 0.002' \\
&\quad restrictionC \quad '2x1 >= 1.57' \\
&\quad \}
\end{aligned}
$$

(Código 6.1)

Fuente: Alejandro Rodas Vásquez

Allí se puede ver claramente cómo las instrucciones por medio de su semántica, funcionan en conjunto alcanzando comunicar el contexto. Ahora bien, analizando la semántica de la instrucción *subjectTo* de forma aislada (es decir, que se realiza el análisis del nombre de la instrucción solamente), esta no logra por sí sola transmitir un significado comunicativo, a diferencia de las instrucciones que posee un lenguaje de propósito general como es el caso de la función *compareTo(String anotherString)*[13] perteneciente a Java, donde haciendo el mismo análisis aislado se logra comprender que el papel de dicha función es realizar una comparación de dos cadenas desde la parte lexicográfica, esto significa que el nombre que posee la función tiene una semántica tal, que logra comunicar su funcionalidad por si sola.

6.1.1 Encadenamiento de métodos (Method Chaining)

El *Encadenamiento de Métodos* (código 6.2), también llamado interfaz fluida (*fluent interface*) es quizás el patrón más utilizado en la implementación de un LEDI [4]. Como su nombre lo dice, esta técnica consiste en realizar invocaciones sucesivas a diferentes métodos de modo que estos se vayan encadenando en una secuencia coherente. Una aproximación conocida a esta técnica se puede encontrar en la implementación del *Patrón de Diseño Constructor*.

```
computer()
    .processor()
        .cores(2)
        .speed(2500)
        .i386()
    .disk()
        .size()
    .disk()
        .size(75)
        .speed(7200)
        .sata()
.end()
```

(Código 6.2)

Fuente: [6]

13 Compara dos cadenas lexicográficamente.
http://docs.oracle.com/javase/7/docs/api/java/lang/String.html#compareTo(java.lang.String)

El contexto de dominio que se plantea en (código 6.2) es la empleo de un LEDI que permite modelar un Computador cualquiera describiéndolo a partir de sus características. Detalle la forma en cómo los métodos son invocados y la secuencia del llamado a estos, permitiendo así que el usuario pueda expresarse de forma natural, utilizando los términos propios del dominio en una secuencia coherente y apegado al dominio.

El *Encadenamiento de Métodos* dentro de su concepto funciona de una forma sencilla; cada uno de los métodos debe retornar un objeto que es del mismo tipo del que ha servido para invocar al método actual.

Es decir, generalmente cuando se define un método (en este caso el método *setSpeed* para definir la propiedad *speed*) se tiene el siguiente fragmento de código:

```
public void setSpeed (int arg){
    this.speed = arg;
}
```

(Cógido 6.3)

Fuente: [6]

Sin embargo, aplicando el patrón señalado, el fragmento anterior toma la siguiente estructura:

```
private HardDrive speed (int arg){
    speed = arg;
    return this;
}
```

(Código 6.4)

Fuente: [6]

Note el empleo de la instrucción *this* que permite tener una referencia al objeto que ha invocado el método *speed,* de esta forma se puede llamar a la próxima función, y así sucesivamente con las otras. Cabe aclarar que estas funciones están definida dentro de las clases que hacen parte del *Modelo de Domino.*

Por otro lado, se puede estar preguntado ¿por qué se cambia el nombre del método de *setSpeed* a *speed*? En efecto, al utilizar el *Encadenamiento de Métodos* se viola una de las buenas prácticas en la codificación, que se sugiere que el nombre de las funciones deben ser significativos (como se hizo mención al inicio de este apartado), sin embargo, recuerde que cada una de estos métodos debe tener un significado en el contexto del dominio y no de forma aislada.

De esta manera, la creación de un método con el nombre *speed* es más acorde con el dominio que se plantea en el ejemplo. No obstante, es importante considerar que una de las desventajas de esta técnica es la proliferación de pequeños métodos que no tengan mucho sentido por sí mismos, de modo que es fundamental realizar un buen análisis del vocabulario que componen el *Dominio*, pues este es la base para la construcción de los métodos que ayudarán a conformar la sintaxis del LEDI.

6.1.2 Función anidada (Nested Function)

```
computer(
    processor(
        cores(2),
        speed(2500),
        i386
    ),
    disk(
        size(150);
    ),
    disk(
        size(75),
        speed(7200),
        SATA
    )
);
```

(Código 6.5)

Fuente: [6]

El patrón de *Función Anidada* se compone de una combinación de funciones que invocan a otras funciones como argumentos de alto nivel [6]. Una de las ventajas que propone este patrón es la facilidad para expresar una estructura jerárquica en la instrucción, a diferencia del *Encadenamiento de Métodos* donde una posible visualización de una jerarquía dentro de las instrucciones solo es permitida mediante la definición de una indentación.

Así pues, una de las ventajas que presenta el patrón de *Función Anidada* es la forma de evaluación de sus instrucciones, donde en primer lugar se evalúan las funciones que sirven como argumentos y por último la función principal que los contiene, de esta forma las funciones que han sido invocadas como parámetros, retornan a la función principal una serie de valores en el formato indicado.

El siguiente fragmento de código presenta un ejemplo de esto:

processor(cores(2), speed(25000), i386()) *(Código 6.6)*

Fuente: [6]

Para esta instrucción se examina primero las funciones que sirven como parámetros (*cores, speed y i386*), las cuales retornarán a la función principal (*processor)* los valores en el formato deseado.

Sin embargo, antes de seleccionar el patrón de *Función Anidada* es conveniente analizar la dirección de lectura de las instrucciones que se desean presentar, tenga en cuenta que al implementar esta técnica el usuario debe iniciar la lectura e interpretación de la instrucción empezando por los argumentos más internos y devolviéndose hasta llegar a la función principal o función padre. Dicha lectura puede estructurarse o tener un orden por medio del empleo de signos de puntuación como paréntesis o comas (como se presenta en un lenguaje de propósito general), sin embargo este enfoque puede funcionar si el LEDI que se está creando tiene como usuario final a personas que se desempeñan en el área de la computación, como lo es un programador, el cual está acostumbrado a este tipo de sintaxis.

Por otro lado, si lo que se desea es ofrecer al usuario un lenguaje que presente una secuencia de instrucciones que se pueda leer de izquierda a derecha las técnicas de *Encadenamiento de Métodos* y *Secuencia de Funciones* son una mejor opción.

6.1.3 Secuencia de funciones (Function Sequence)

```
computer();
    processor();
        cores(2);
        speed(2500);
        i386();
    disk();
        size(150);
    disk();
        size(75);
        speed(7200);
        sata();
```

(Código 6.7)

Fuente: [6]

A diferencia del *Encadenamiento de Métodos* y *Función Anidada,* el patrón de *Secuencia de Funciones* produce una serie de llamadas sin relación entre sí [6]. Al contrario de los anteriores patrones que requieren un cierto intercambio de datos entre las funciones que componen la instrucción, en el patrón de *Secuencia de Funciones* no existe una relación de datos entre ellas, por lo que se necesitan una serie de técnicas que permitan realizar el análisis de la secuencia de instrucción, capturando el dato que está manipulando la función actual y guardándolo en una variable que contenga el contexto requerido durante el análisis para ser transferido posteriormente a la siguiente función que conforma la instrucción expresada; esta técnica es llamada *Variable de Contexto (Context Variable)* [6].

Sin embargo, a pesar de usar esta técnica el empleo de *Secuencia de Funciones* es el patrón menos útil en la combinación de llamadas a funciones para utilizar en un LED, ya que al utilizar una *Variable de Contexto* para efectuar el seguimiento del proceso de análisis es en realidad una tarea demandante, la cual puede llevar a construir un código que es difícil de comprender y propenso a errores.

Por otra parte, pueden existir situaciones que demanden del empleo de este patrón, en las cuales se requiere describir o modelar un contexto a través múltiples declaraciones (funciones de alto nivel) que *no necesariamente deban estar fuertemente relacionadas*. En estos casos solo sería necesaria una sola *Variable de Contexto,* lo cual permitiría mantener el análisis de las instrucciones del LED de forma sencilla.

Por el contrario, si lo que se desea obtener es una serie de expresiones que *si estén relacionadas y puedan describir un contexto en forma estructurada,* los patrones recomendados son *Funciones Anidadas* o *Encadenamiento de Métodos.*

6.1.4 Selección de un patrón de construcción para implementar en el proyecto

Como se ha podido observar, las anteriores técnicas ofrecen distintos caminos a la hora de implementar la sintaxis que ofrecerá el LED en construcción. Así mismo, haciendo un análisis de la forma y la expresividad a la que se quiere llegar en el proyecto (como se muestra en código 6.8) es conveniente analizar una cuarta técnica presentada en por Martin Fowler [6] llamada *Cierres Anidados (Nested Closures).*

$$max('2x1 + 0.34x2').subjectTo\ \{$$
$$restrictionA \quad '0.03x1 > 0.98'$$
$$restrictionB \quad '10.6x1 <= 0.002'$$
$$restrictionC \quad '2x1 >= 1.57'$$
$$\}$$

(Código 6.8)

Fuente: Alejandro Rodas Vásquez

Esta técnica se adapta perfectamente al uso de Ruby como lenguaje anfitrión y se fundamente en el uso de *Bloques* (ampliamente abortados en el capítulo *Bloques (BLOCKS), Procs y Lambdas*) como un elemento que contribuye al enriquecimiento de la sintaxis.

La idea fundamental de los *Cierres Anidados* es similar al presentado en las *Funciones Anidadas,* con la diferencia que las funciones que son pasadas como parámetros a la función padre esta vez se alojan en el cuerpo del *Bloque.* Para comprender mejor este concepto observe el siguiente fragmento de código (tenga en cuenta que el siguiente ejemplo es implementado utilizando Ruby como lenguaje).

```
processor(cores 2, i386)
```

(Código 6.9)

Fuente: [6]

Allí se puede destacar la utilización de la técnica de *Funciones Anidadas* donde la función padre es llamada *processor* y las funciones hijas *cores* y *i386* han sido pasadas como argumentos. Ahora observe cómo cambia dicha expresión empleando la técnica de *Cierres Anidados.*

```
processor do
    cores 2
      i386
end
```

(Código 6.10)

Fuente: [6]

En dicha expresión se tiene la misma función padre, pero en vez de pasar dos *Funciones Anidadas* como argumentos se tiene un solo argumento que es un *Cierre Anidado,* el cual está compuesto de dos expresiones que son *Funciones Anidadas.* De esta forma, la función *processor* es evaluada primero y el *Bloque* es analizado en el momento que se indique, usualmente el análisis de dicho *Bloque* se realiza dentro de la función que lo ha invocado (en este caso la función *processor*), donde para efectos de la presente investigación se ha hecho uso de la *Metaprogramación* para tal fin.

Cabe recordar que una de las limitaciones que existen en la implementación de esta técnica radica en que el lenguaje anfitrión seleccionado suporte el uso de *Bloques*, de esta forma Ruby se presenta como candidato. Así mismo, se encuentra que el patrón *Cierre Anidado* se presenta como una mejora de *Funciones Anidadas, Secuencia de Funciones y Encadenamiento de Métodos [6],* ya que permite combinar estas técnicas creando un lenguaje versátil, con la habilidad de controlar el momento en el cual se realiza la evaluación de los argumentos, como anteriormente se mencionaba.

Por consiguiente, se hace elección de la presente técnica para ser la utilizada en la creación del LEDI objeto de la investigación. De esta forma, una vez realizada la selección del patrón a implementar y evaluado el aporte que este ofrece a la expresividad de la sintaxis del lenguaje al momento de expresar el modelado de un problema de optimización, se está ya en condiciones de iniciar la construcción de la arquitectura en la que se sustentará el LEDI, el cual es precisamente el objetivo del próximo capítulo.

7. DEFINICIÓN DE LA ARQUITECTURA UTILIZANDO LENGUAJE DE MODELADO UNIFICADO (UML), EN LA IMPLEMENTACIÓN DE UN LENGUAJE ESPECÍFICO DE DOMINIO INTERNO

En este capítulo se mostrará el proceso de construcción de la arquitectura del LEDI. Se podrá observar el proceso de construcción de la arquitectura empleando la metodología C4 (anteriormente referenciada) y al mismo tiempo se revelará no solo la perspectiva técnica (como es la definición de los diagramas UML) sobre la cual se cimentó la arquitectura sino que se enunciarán los problemas que surgiendo a medida que se fue creando la misma, y cómo estos fueron superados mediante la propia arquitectura.

7.1 CREACIÓN DE UN DIAGRAMA DE CONTEXTO DEL SISTEMA PARA UNA VISTA GENERAL DEL LENGUAJE ESPECÍFICO DE DOMINIO INTERNO (LEDI)

Uno de los objetivos en aplicar la metodología C4, es mejorar la abstracción en la representación del sistema mediante la creación de varios diagramas que representen diferentes niveles de dicha abstracción. Por lo tanto, iniciar la construcción de la arquitectura con un *Diagrama de Contexto del Sistema* (que puede ser equivalente a un *Diagrama de Casos de Uso*) obtener un vista del sistema desde nivel abstracción general visualizándolo como un solo bloque.

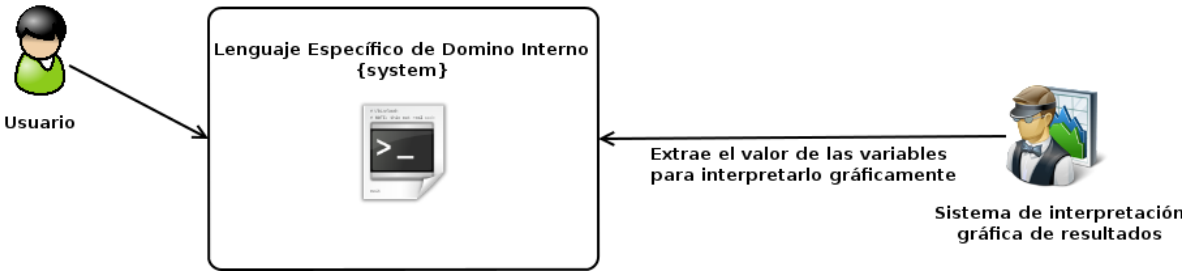


Fig. 10. Diagrama de Contexto del Sistema (Fuente: Alejandro Rodas Vásquez)

En Fig. 10 se puede notar dos actores:

- ***Actor Usuario:*** Este actor juega el papel de *Usuario* del LEDI he interactúa con los casos de uso *Modelado de Problema de Optimización* y *Obtener los valores de las variables de decisión.* Donde el primero, se enfoca en la construcción de los componentes de la arquitectura que le permitirán al actor utilizar la infraestructura que soporta el *modelo de dominio* del LEDI; y el segundo caso que ha sido diseñado para ejecutar todos aquellas

operaciones que implican la resolución del modelado de la función a optimizar, entre estas la conexión entre el *algoritmo de resolución de problemas de optimización* creado en C y el propio LED implementado en Ruby.

- ***Actor Sistema de Interpretación Gráfica de Resultados:*** Aunque este actor no tiene presencia obvia en la presente investigación, se ha tomado la decisión en incluirlo en el modelo puesto que tomando en consideración trabajos futuros es importante contar con un sistema que permita visualizar los resultados obtenidos de forma gráfica. Esta consideración se verá reflejada en la arquitectura como se observará más adelante.

7.2 CREACIÓN DE UN PATRÓN DE ARQUITECTURA EN CAPAS PARA LA CONSTRUCCIÓN DEL LEDI

Continuando con la aplicación de la metodología C4, una vez comprendido cómo el sistema interactúa en el contexto en el cual se está presentado, el siguiente paso es representarlo por medio de un diagrama que permita analizarlo en elementos con una funcionalidad especializada.

Así pues, la perspectiva a crear es una visualización del sistema por medio de lo que han sido llamados *Contenedores* (*Containers*), los cuales se representarán por medio de un *Diagrama de Paquetes*. Esta vista permite descomponer el *Diagrama de Contexto del Sistema* en un conjunto de bloques que tendrán una comunicación entre ellos y responsabilidad individuales asignadas.

De esta forma, para la construcción de la arquitectura que soportará el LEDI se tomó como base la utilizada por AMPL (A Mathematical Programming Language)[14]. Dicha arquitectura inicialmente presenta dos capas, el *Modelador* y el *Solver*, donde según [16] estas se definen como*:*

- ***Modelador:*** Está conformado por el modelo expresado desde la concepción matemática.
- **Solver[15]:** Programa que contiene los algoritmos que resuelven los distintos problemas.

14 Es un lenguaje de modelado algebraico para la programación matemática; fue diseñado e implementado por Robert Fourer, David M. Gay y Brian W. Kernighan a mediados de 1985 [14].

15 Es un programa que contiene un algoritmo para resolver problemas de optimización [14].

La capa *Modelador* (nombrada como *DSL Model*) contiene el *Modelo de Dominio* del LEDI, de esta forma el usuario al modelar el problema de optimización está interactuando con esta capa directamente. Así mismo, cabe mencionar que la totalidad de la misma será construida utilizando Ruby.

Por otro lado, la capa *Solver* (nombrada como *C Solver*[16]) contiene los algoritmos que permiten resolver el problema de optimización; a diferencia de la capa anterior, esta será implementada utilizando C.

De esta forma, se tiene una arquitectura de dos capas. Sin embargo tener un sistema con esta característica presentaría un problema de fuerte acoplamiento, sumando a este inconveniente el hecho que dichas capas son construidas en tecnologías diferentes. Por consiguiente, se observa la necesidad de incorporar una capa intermedia que permita la comunicación entre ellas, teniendo como resultado una arquitectura de tres capas (Fig. 11) la cual sustenta la funcionalidad del LEDI.

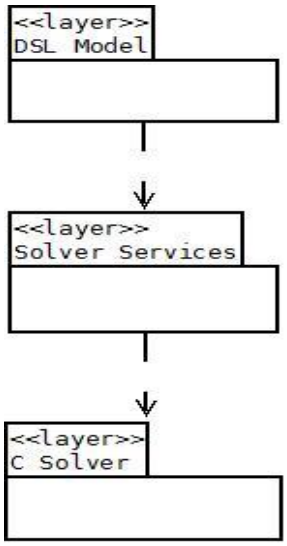


Fig. 11. Arquitectura de tres capas que sustenta el LEDI (Fuente: Alejandro Rodas Vásquez)

16 Para un caso práctico se ha implementado en lenguaje C el algoritmo de Garkinkel.

A continuación se presentan las funcionalidades que desempeñan cada una de las capas que conforman la arquitectura:

- **Capa DSL Model:** Esta capa contiene las estructuras que componen el Modelo de Dominio. Estas estructuras son representadas mediante clases, donde por medio de sus atributos y métodos el usuario posee los elementos sintácticos provistos por el LEDI necesarios para la representación del modelo a optimizar. Del mismo modo, esta capa posee un componente dedicado a albergar los casos de prueba que ayudan a comprobar el Modelo de Dominio y los nuevos elementos que se añadan a este.
- **Capa SolverServices:** Esta capa permite la comunicación entre las capas *DSL Model*y *C Solver*,las cuales están construidas en los lenguajes de programación Ruby y C, respectivamente.Para solventar este problema entre plataformas, se ha hecho uso de la librería FFI (*Foreign Function Interface*[17]), la cual permite hacer llamados desde Ruby a funciones implementadas en C.Por lo tanto, la capa *DSL Model* puede llamar las rutinas algorítmicas alojadas en la capa *C Solver*[7].
- **Capa C Solver:** Esta capa contiene los algoritmos que permiten resolver problemas de optimización, donde como ya se ha mencionado, estos están construidos en el lenguaje de programación C.

7.3 DESCRIPCIÓN DE LA INTERACCIÓN ENTRE LAS CAPAS QUE CONFORMAN LA ARQUITECTURA A TRAVÉS DE SUS COMPONENTES

Una vez obtenida la representación de la arquitectura del sistema por medio de un *Diagrama de Contenedores,* el siguiente paso es descomponer cada *Contenedor* en bloques que han sido llamados Componentes (*Components*). Tal pues, como se menciona en [27] un *Contenedor* representa el lugar en el cual los *Componentes* son ejecutados.

En Fig. 12 se muestra el sistema por medio de un *Diagrama de Componentes,* y cómo cada capa representada (por su correspondiente Contenedor) está conformada por uno o varios *Componentes*.

En el nivel superior, la capa *DSL Model* consta de tres componentes:

- ***Componente DSL Model:*** Contiene las estructuras que conforma el *Modelo de Dominio,* es decir, el LEDI propiamente dicho. Por lo tanto, es este *Componente* el que interactúa directamente con el actor *Usuario*
- ***Componente Test DSL Model:*** Dedicado a los casos de prueba y tiene como finalidad comprobar el Modelo Semántico residente en el Modelo de

17 Librería para Ruby creada por Wayne Meissner.

Dominio, de modo que cualquier modificación o evolución de dicho modelo sea comprobable y validada a través de dichas pruebas o *test.*

- ***Componente Builder*:** Es el encargado de hacer el llamado a los métodos que yacen en el componente *SolverServices* (el cual está alojado en el *Contenedor* del mismo nombre).

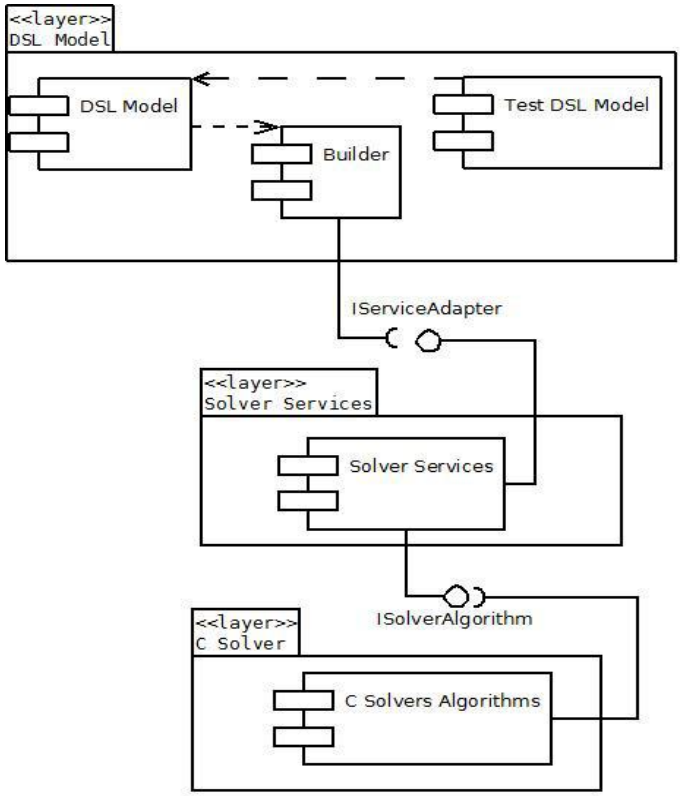


Fig. 12. Diagrama de componentes (Fuente: Alejandro Rodas Vásquez)

Por otro lado, como se puede observar en la figura los componentes *Builder* y *SolverServices* hacen uso de la interfaz *IserviceAdapter.* Esta interfaz no solo cumple el objetivo de proporcionar el desacoplamiento y flexibilidad entre capas, sino que también forma parte en la implementación del *Patrón de Diseño Adaptador.* Mediante este patrón, se logra que los resultados obtenidos por el componente *SolverServices* por medio del llamado a las rutinas implementadas por el componente *C SolverAlgorithms*, puedan ser almacenados en arreglos tipo *Hash*[18], de este modo el componente *Builder* recibe los resultados en un formato

18 Es una estructura de datos que mantiene un conjunto de objetos conocidos como *llaves*, y asocia un valor a cada uno de ellas.

estándar no importando el tipo de algoritmo de optimización ejecutado por el componente *C SolverAlgorithms*.

De la misma manera, la interfaz *ISolverAlgorithm* permite el desacoplamiento entre las capas *SolverServices* y *C Solver*. Sin embargo, es esta interfaz la encargada de implementar la librería FFI, de esta forma se logra la interconexión entre las dos tecnologías que conforman la arquitectura.

7.4 DESCOMPOSICIÓN DE LOS COMPONENTES QUE CONFORMAN LA ARQUITECTURA A TRAVÉS DE UN DIAGRAMA DE CLASES

7.4.1 Descripción de los componentes básicos

Una vez descrita la arquitectura por medio de un *Diagrama de Contenedores* y posteriormente por un *Diagrama de Componentes*, y siguiendo la metodología propuesta para la implementación del LEDI, se presenta a continuación el nivel de descripción más detallado de la misma por medio de un *Diagrama de Clases*, tal como se plantea en [37].

De igual forma, ya que el enfoque de implementación escogido para la construcción del LEDI radica en la combinación de los patrones *Encadenamiento de Métodos* y *Cierre Anidado (Nested Closures)*, las características funcionales del lenguaje yacen en la creación de un *Modelo de Dominio* que por medio de la definición de clases y el empleo de sus métodos como elementos que provee la sintaxis del LEDI, es conveniente analizar la arquitectura desde esta perspectiva de un *Diagrama de Clases*.

Como se puede observar en Fig. 13, *OptimizationFunction* y *Restriction* son clases que están asociadas por una relación de *Composición*; son estas clases las que conforman el *Modelo de Dominio* de la arquitectura.

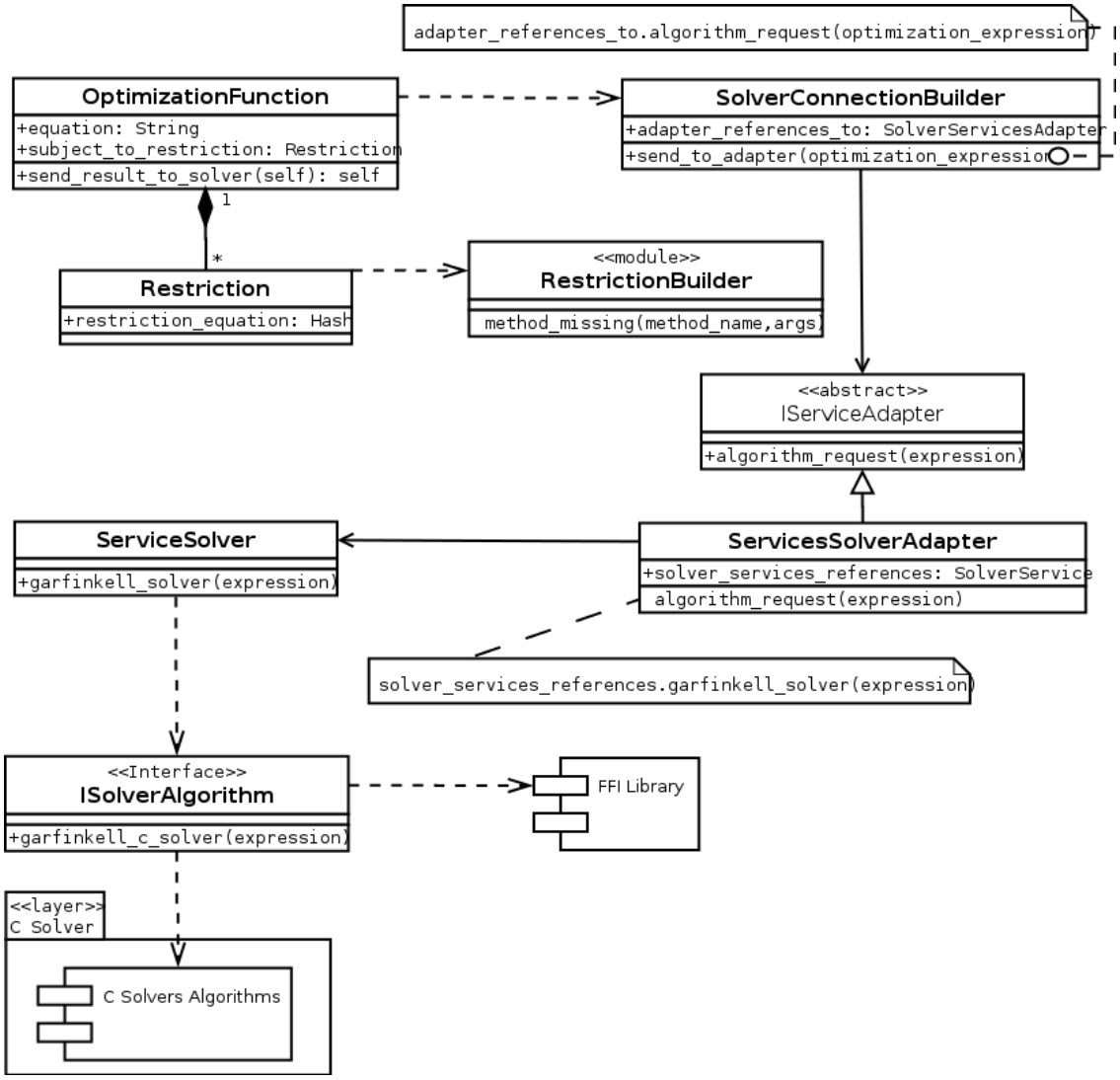


Fig. 13. Diagrama de Clases que describen los componentes DSL Model, Builder y Solver Services (Fuente: Alejandro Rodas Vásquez)

Se puede notar que la clase *OptimizationFunction* posee dos atributos que son: *equation*(permite almacenar la ecuación que conforma la función objetivo) y *subject_to_restriccition* (permite almacenar en forma de objeto la serie de restricciones a las que está sujeta la función a optimizar), este último atributo está relacionado con la clase *Restriction*, la cual posee el atributo llamado *restriction_equation* correspondiente a una estructura de tipo *Hash* destinada a almacenar las restricciones relacionadas con la función de optimización, donde la *llave* del arreglo *Hash* es el nombre de la restricción y el *valor* de dicha *llave* es la inecuación que expresa la restricción.

Como se ha mencionado durante el documento, toda la sintaxis que posee el LEDI yace en su *Modelo de Dominio* y esta es representada por los métodos que yacen en este. Sin embargo, ya que toda restricción lleva un nombre que la identifica es difícil saber cuál será el asignado por el usuario a cada restricción de esta forma haciendo imposible crear un método que sea residente permanente en el *Modelo*.

Por tanto, esta dificultad es sorteada mediante la implementación del módulo *RestrictionBuilder,* el cual ha sido diseñado con el propósito de ser el encargado de construir mediante la técnica de *Generación Dinámica de Métodos,* todas aquellas restricciones que el usuario ingrese. De esta forma, como se puede evidenciar en Fig. 13 se define la relación entre dicho módulo y la clase *Restriction.* A continuación se muestra el código implementado donde se puede observar la definición de las clases que conforman el *Modelo de Dominio* y la invocación del módulo *RestrictionBuilder* la clase *Restriction.*

```
class OptimizationFunction

 attr_accessor :equation, :subject_to_restriction

 def initialize equation
  @equation = equation
 end

 def self.create_object equation
  funcion_optimizacion = new equation
  funcion_optimizacion
 end

 def subject_to &block
  restriction = Restriction.new
  restriction.instance_eval(&block)
  @subject_to_restriction = restriction
  self
 end
 alias_method(:subjectTo, :subject_to)
 alias_method(:s_t, :subject_to)

 def send_result_to_solver
  result = SolverConnectionBuilder.new.send_to_adapter(self)
  self
 end

 def to_s
  "(#{@equation}, #{@subject_to_restriction.restriction_equation})"
 end

 class Restriction
  attr_accessor :restriction_equation

  include RestrictionBuilder
  def initialize
   @restriction_equation = {}
  end

  def restriction_number?
   @restriction_equation.size
  end
 end

end
```

(Código 7.1)

Fuente: Alejandro Rodas Vásquez

Por otro lado, al observa en el Anexo F se puede notar empleo del atributo *restriction_equation* en la implementación del módulo *RestrictionBuilder* logrando así el almacenamiento del nombre asignado por el usuario a la restricción como *llave* del arreglo *Hash* y la inecuación de dicha restricción como *valor* de su correspondiente *llave*, como anteriormente se ha mencionado.

Así pues, una vez dada la estructura apropiada al objeto de tipo *OptimizationFunction* (código 7.1) el cual representa el modelado del problema de optimización con sus respectivas restricciones, el usuario debe hacer uso del método *send_result_to_solver,* el cual permite enviar dicho objeto hacia la clase *SolverConnectionBuilder* (código 7.2), la cual es la encargada de entablar la conexión con la capa *C Solver,* esto se hace invocando la implementación del método *algorithm_request*, la cual reside en la clase *ServiceSolverAdapter* y se define en la clase abstracta *IserviceAdapter.*

De este modo se inicia el proceso que implica enviar el modelo de optimización hacia los *algoritmos de resolución* mediante el método *send_to_adapter* el cual es llamado dentro del método *send_result_to_solver*, como se puede observar en (código 7.1)

```
require_relative '../services_solver/services_solver_adapter'

class SolverConnectionBuilder

  def send_to_adapter optimization_expression
    adapter_references_to = ServicesSolverAdapter.new
    adapter_references_to.algorithm_request(expression_model)
  end

end
```

(Código 7.2)

Fuente: Alejandro Rodas Vásquez

Cabe recordar que uno de los objetivos que busca esta arquitectura es permitir que el LEDI puede ejecutar diversos tipos de algoritmos de resolución de problemas de optimización, por tal razón se ha creado la capa *C Solver* como se mencionó anteriormente. Sin embargo, existe el inconveniente que no todos estos algoritmos presentan una salida estándar, es decir, un formato único estructurado para el resultado.

Por otro lado, también se requiere un lugar dentro de la arquitectura donde residan estos resultados, de modo que sea posible para un sistema externo (como por ejemplo, un sistema de interpretación gráfica) consultarlos y extraerlos para hacer uso de ellos. Por tal motivo, sea escogido el *Patrón de Diseño Adaptador* como un medio factible para resolver dicho inconveniente.

Así mismo, para explicar cómo se implementa este patrón dentro de la arquitectura diseña, a continuación se realizará una analogía entre las clases residentes en la arquitectura y las presentadas en el siguiente diagrama UML (Fig. 14)

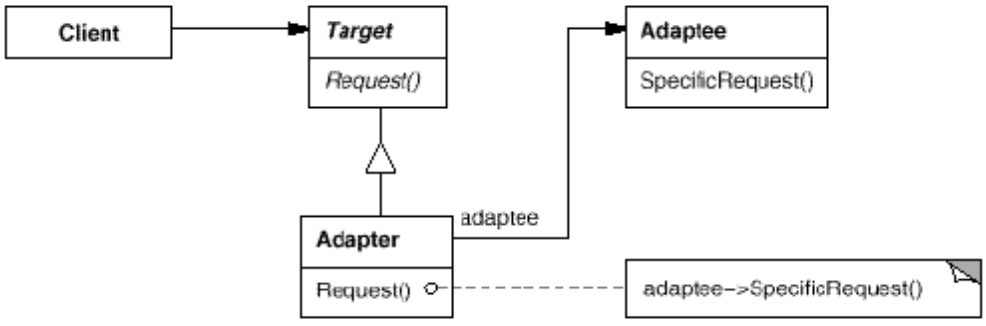


Fig. 14. Diagrama de implementación del Patrón de Diseño Adaptador [37].

Para implementar este patrón se requiere una clase *Target,* esta clase tiene como propósito ofrecer las interfaces que serán utilizadas por la clase *Client.* Esta clase *Target* se implementa como la clase abstracta *IServiceAdapter* la cual ofrece el método *algorithm_request* que es utilizado por la clase *SolverConnectionBuilder* (la cual realiza el papel de clase *Client*) en la implementación del método *send_to_adapter,* el cual tiene la tarea de enviar el objeto que representa la función de optimización (junto con sus correspondientes restricciones), hacia la clase *ServicesSolverAdapter*.

Por otro lado, la clase *Adapter* que es representada en la arquitectura por la clase *ServiceSolverAdapter,* esta permite la interacción entre la clase *Client* (*SolverConnectionBuilder*) y la clase *Adaptee* (*ServiceSolver*), puesto que esta implementa el método *algorithm_request,* el cual es el encargado de realizar el llamado a la rutina que contienen los algoritmos de optimización implementados en C mediante la invocación del método *garfinkell_solver*, propio de *ServiceSolver* (código 7.3).

Por consiguiente, esto permite que todas las respuestas arrojadas por los algoritmos de resolución sean captadas por *ServiceSolverAdapter* (código 7.4) y que dentro del método *algorithm_request* se puedan procesar dichos resultados y así crear una salida estándar con un formato único estructurado, *de forma que puedan ser visualizados por el usuario ya sea por pantalla o por medio de una aplicación orientada para tal fin,* como anteriormente se mencionó.

```
class ServiceSolver
  include ISolverAlgorithm

  def garfinkell_solver
    garfinkell_c_solver
  end
end
```

*(*Código *7.3)*

Fuente: Alejandro Rodas Vásquez

```
require_relative 'service_solver'

class ServicesSolverAdapter
 attr_accessor :services_solver_references

 def initialize
  @services_solver_references = ServiceSolver.new
 end

 def algorithm_request expression
   #In this method you can create a format to 'expression'
       #garfinkell_solver method is a which connect to FFI library
   @services_solver_references.garfinkell_solver
 end
end
```

(Código 7.4)

Fuente: Alejandro Rodas Vásquez

Por otro lado, se tiene la interfaz IsolverAlgorithm (código 7.5) la cual tiene como tarea acceder al componente C Solver que contiene las rutinas implementadas en C, donde para lograr este fin depende de la librería FFI.

```
require 'ffi'

module ISolverAlgorithm
 extend FFI::Library

 #directory path optimization algorithms are
 ffi_lib 'c_solvers/solvers_algorithm.so'
 attach_function :garfinkell_c_solver,[],:int
end
```

(Código 7.5)

Fuente: Alejandro Rodas Vásquez

Esta librería es invocada por medio de la instrucción *require 'ffi'*. Por otro lado, la sentencia *ffi_lib* permite indicar la ruta en donde se encuentra el archivo que actúa como librería dinámica *(solvers_algorithm.so),* es allí donde están alojados los algoritmos de resolución para los problemas de optimización; que para efectos de la presente investigación se ha escogido el algoritmo de Garfinkell como un método para abordar problemas que abordan el área de la *Programación Lineal* (cabe aclarar que la implementación de este algoritmo no está dentro de los objetivos de la investigación, pero será considerado para trabajos futuros).

Para lograr realizar el llamado de dichos algoritmos se utiliza la función *attach_function* (propia de la librería *FFI*), donde el primer parámetro es el nombre de las funciones que se encuentran en la librería *solvers_algorithm.so* y es en ella donde se encuentra el algoritmo requerido para la resolución del problema a optimizar.

7.4.2 Integración del componente *ExpressionParser* para el análisis sintáctico de expresiones matemáticas

En esta instancia se ha descrito el comportamiento que presenta la arquitectura que sustenta el LEDI por medio de la interacción que existe entre sus componentes. Sin embargo, en este punto de la implementación de la arquitecta el lenguaje no posee ningún medio por el cual detecte si el problema de optimización a modelar es del tipo *Programación Lineal* o *Programación No Lineal*, de esta forma es necesario agregar a la infraestructura ya descrita, un componente que permita analizar cada una de las expresiones matemáticas (manifestadas en la función de optimización y sus restricciones) para posteriormente determinar en qué categoría se encuentra el problema que se está modelando, de esta manera el LEDI podrá utilizar el algoritmo de resolución apropiado.

Por tanto, la capa *DSL Model* se modifica agregando el componente llamado *ExpressionParser* (Fig. 15)*,* el cual tiene como objetivo realizar dicho análisis de las expresiones matemáticas, donde este interactúa directamente con el componente *Builder.*

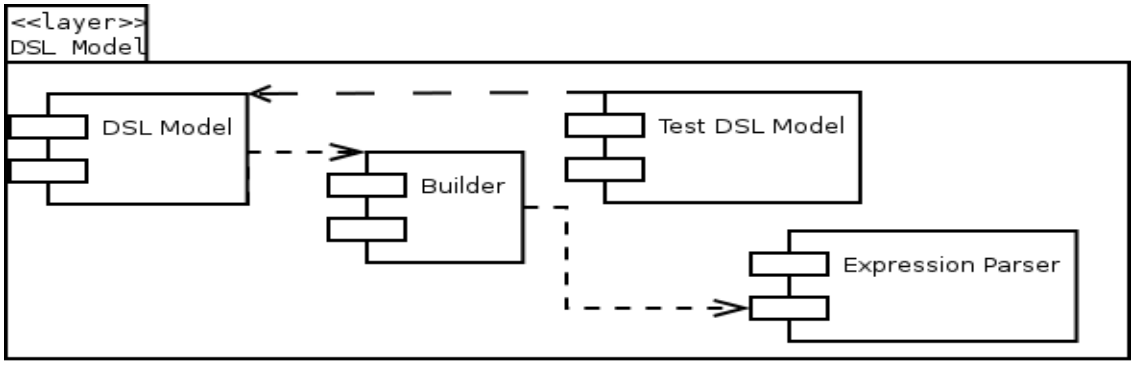


Fig. 15. Componente *ExpressionParser* (Fuente: Alejandro Rodas Vásquez)

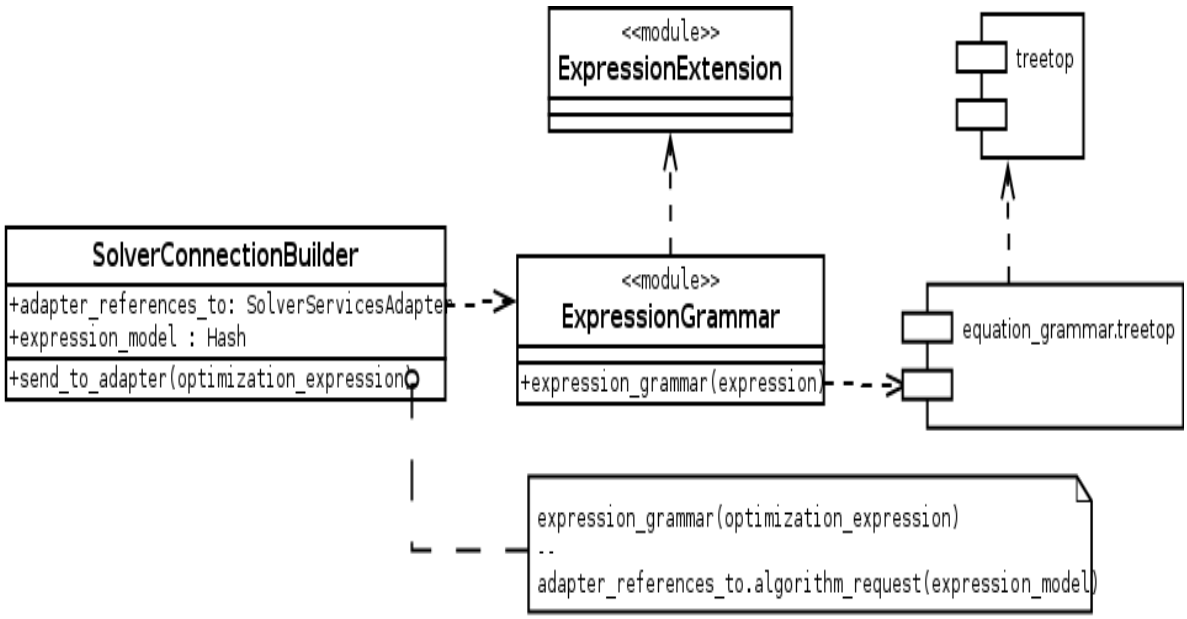


Fig. 16. Módulos que conforman el componente *ExpressionParser* y su interacción con la clase *SolverConnectionBuilder* (Fuente: Alejandro Rodas Vásquez)

Del mismo modo, este nuevo componente es conformado por los módulos *ExpressionExtensio*nes y *ExpressionGrammar* (este utilizado por la clase *SolverConnectionBuilder*), el cual utiliza el archivo de configuración *equation_grammar.treetop* junto con la librería *Treetop* (Fig. 16) la cual debe ser instalada en el ambiente de desarrollo pues ofrece todas aquellas funcionales que permitirán realizar el análisis sintáctico de la expresión matemática.

Precisamente, durante el proceso de investigación del proyecto surgió la necesidad (como ya se menciono) de lograr identificar y categorizar el tipo de ecuaciones que el usuario está utilizando en el modelado del problema de optimización, esto con el fin de permitir que el LEDI pueda invocar internamente el algoritmo de resolución apropiado para generar el resultado esperado. Adicionalmente, también surgió la necesidad extraer y separar de la propia expresión las variables y coeficientes que la conforman, de modo que puedan ser entregados en un formato establecido a la clase *ServiceSolverAdapter*, logrando así la aplicación el *Patrón de Diseño Adaptador* (explicado anteriormente).

Por consiguiente, realizar esta separación fue necesario ya que realizar dicho análisis haciendo un uso básico de una *expresión regular* no ofrecería la funcionalidad y versatilidad requerida. De esta forma, se necesitaba una herramienta que permitiera definir la sintaxis de la expresión matemática y realizara el análisis requerido, así como también que fuese compatible con Ruby. Tomando en cuenta estas consideraciones se escoge y se implementa el analizador sintáctico llamado *Treetop* para ser parte de la arquitectura del lenguaje.

Como primer paso es imperativo definir la *estructura gramatical* con sus correspondientes reglas en un archivo que posea la extensión *treetop,* en este caso este es llamado *equation_grammar.treetop*. A continuación se muestra la estructura creada para el lenguaje en construcción.

```
grammar ExpressionGrammar

  rule equation
   (number /  identifier / equality_inequality)* <ExpressionExtension::EquationLiteral>
  end

  rule expression
   number identifier? <ExpressionExtension::ExpressionLiteral>
  end
```

(Código 7.6)

```
  rule number
   ('+' / '-')?  [0-9]+ ('.' [0-9]+)? <ExpressionExtension::NumberLiteral>
  end

  rule identifier
    [a-zA-Z]+ ([0-9]+)? ('^' [0-9]+)?<ExpressionExtension::IdentifierLiteral>
  end

  rule equality_inequality
    ( '=' /  '>='  / '<=' / '<' / '>')+  <ExpressionExtension::OperadorLiteral>
  end

end
```

Fuente: Alejandro Rodas Vásquez

Se puede observar que al inicio del archivo se utiliza la instrucción *grammar,* la cual se emplea para indicar que se introduce una nueva gramática que tiene como nombre *ExpressionGrammar,* este será utilizado dentro de la arquitectura para lograr hacer el llamado del analizador de expresiones.

Del mismo modo, se observa dentro de este archivo las expresiones regulares que procesaran la expresión, sin embargo, puede dar la impresión que ellas están contenidas dentro de algún tipo de función. No obstante, esta es la forma de definir una *estructura gramatical* en *Treetop*. Para lograr esto, es necesario emplear la instrucción *rule*, la cual indica que se está definiendo un *regla de análisis* (*parsing rule*) dentro de la gramática; dicha regla debe tener un nombre que la identifique. Esta es una de las características principales que presenta *Treetop,* pues permite crear *reglas* haciendo referencia a otras, a través de su nombre, como se puede notar al inicio de (código 7.6), donde la *regla* que lleva el nombre de *equation* está compuesta por las reglas con el nombre *number, identifier y equality_inequality*.

De esta forma, cada vez que se analiza una expresión esta es analizada mediante las *reglas* que se han definido, generando los *nodos* que conforman el *árbol de análisis sintáctico.*

Por otro lado, una característica importante que ofrece *Treetop* es su enfoque orientado a objetos, esto quiere decir que no solamente se limita a evaluar la correcta composición de la cadena a analizar, sino que cada uno de los nodos perteneciente al *árbol de análisis sintáctico* es representado como un objeto de la clase *Treetop::Runtime::SyntaxNode*, de esta forma es posible asociar métodos que contengan cierta lógica a dichos *nodos,* permitiendo realizar algún tipo de operación a la expresión que se procesa en dicho nodo.

En (código 8.6) se puede observar dicha asociación. Note que cada expresión regular es seguida por una referencia a una clase que está alojada en el módulo llamado *ExpressionExtension*, donde dicha referencia se hace empleando los caracteres *paréntesis angulares (<>).* Esto significa que el comportamiento asociado a cada una de las reglas definidas está establecido en este módulo, el cual se muestra en (código 7.7).

```
module ExpressionExtension
  class EquationLiteral < Treetop::Runtime::SyntaxNode
    def content
      elements.map { |node| node.content   unless node.class.name == "Treetop::Runtime::SyntaxNode" }
    end
  end

  class ExpressionLiteral < Treetop::Runtime::SyntaxNode
    def content
      self.text_value
    end
  end

  class NumberLiteral < Treetop::Runtime::SyntaxNode
    def content
      self.text_value
    end
  end

  class IdentifierLiteral < Treetop::Runtime::SyntaxNode
    def content
      ["^"] == self.text_value.scan(/\^/)) ? self.text_value.to_sym : self.text_value
    end
  end

  class OperadorLiteral < Treetop::Runtime::SyntaxNode
    def content
      self.text_value
    end
  end

end
```

(Código 7.7)

Fuente: Alejandro Rodas Vásquez

Observe la forma en cómo están definidas las clases que son referenciadas por cada una de las reglas que se muestran en (código 7.6), así mismo cada clase hereda de *Treetop::Runtime::SyntaxNode* permitiendo de esta forma asociar el o los métodos que están en dichas clases a la regla que la invoca. Por consiguiente, se concluye que cada regla tiene asociada un método llamada *content.*

Es en esta instancia donde por medio de dicha funcionalidad se logra determinar si la expresión matemática que se está analizando es o no lineal. De forma que, una vez finalice el análisis, la arquitectura del LEDI podrá determinar a cual algoritmo de resolución se debe enviar el problema de optimización modelado.

Para comprender mejor cómo se realiza este proceso, observe la clase denominada *IdentifierLiteral (*en código 7.7*)* la cual está asociada a la regla *identifier* (en código 7.6). Esta clase define el método *content*, el cual posee una lógica distinta a los que se encuentran alojados en las otras clases.

Esto significa que el nodo que se identifica con el nombre de regla *identifier,* al invocar el método *content* puede ejecutar el código que este posee en el mismo instante en el que se está realizando el análisis sintáctico de la expresión y que concuerde con el patrón de la *expresión regular* definida en dicha regla.

El objetivo de la regla *identifier* es definir un patrón que identifique si una variable (ya sea de la función de optimización o de la restricción indistintamente), al utilizar el carácter ^ pueda indicar si está elevada a una potencia. Sin embargo, un regla por si sola lo único que realiza es la comparación entre la *expresión regular* que contiene y la cadena que debe ser analizada, por tal motivo regla *identifier* necesita ser asociada con otro componente que posea cierta lógica y que permita determinar si se está caracterizando un expresión *lineal o no lineal.* Precisamente, este es la funcionalidad que se busca al asociar una regla con una clase que contenga una serie de métodos que describan un comportamiento específico, en este caso la clase *IdentifierLiteral* con su método *content (*código *7.8).*

```
class IdentifierLiteral < Treetop::Runtime::SyntaxNode
  def content
    (["^"] == self.text_value.scan(/\^/)) ? self.text_value.to_sym : self.text_value
  end
end
```

*(*Código *7.8)*

Fuente: Alejandro Rodas Vásquez

Básicamente el comportamiento que manifiesta este método consiste en obtener el fragmento de cadena (mediante la instrucción *self.text_value*) que está siendo procesada en ese momento por el nodo *identifier*, y mediante el empleo de la función *scan*[19] analizar este fragmento en búsqueda del carácter exponente (^). Si este carácter es encontrado quiere decir que la expresión analizada *no es lineal,* y es convertida en *símbolo* mediante la instrucción *self.text_value.to_sym.* Esta conversión es utilizada dentro de la arquitectura como una forma para identificar si se debe emplear un algoritmo de resolución enfocado a problemas *lineales o no lineales*, dicha decisión es realizada por el método *algorithm_request* implementado en la clase *ServicesSolverAdapter*.

Para comprender esta dinámica, suponga que se tiene la expresión matemática *3x^2+6y<=34.45,* esta debe ser analizada de modo que la arquitectura determine qué tipo de problema es.

19 Escanea a través de la cadena en búsqueda de algún carácter que coincida con la expresión regular que recibe como parámetro [31].

Una vez esta expresión es analizada por cada una de las reglas establecidas dentro del *analizador sintáctico* (código 7.6) y ejecutadas las funciones asociadas a ellas (código 7.7) el resultado es la creación del siguiente arreglo (código 7.9).

```
Parser Expression: {:optimization_function=>["3", :"x^2", "+6", "y", "<=", "34.45"]}
```

*(*Código *7.9)*

Fuente: Alejandro Rodas Vásquez

Como puede observar, esta es una estructura de datos tipo *Hash* que ha sido llamada *optimization_function* la cual contiene todos los elementos que conforman la expresión. Note que solamente el segundo elemento del arreglo (*:"x^2"*) es antecedido por el carácter dos puntos (:) esto indica que este es un *Símbolo,* de modo que el método (*content*) asociado con la regla (*identifier*) que define la expresión regular que se encarga de analizar este patrón ha sido ejecutado, como se explicó anteriormente.

Una vez se tiene el arreglo, este es pasado como parámetro al método *algorithm_request, donde se detecta si existe algún elemento de tipo símbolo dentro del arreglo, si es así, se envía un mensaje de advertencia explicando que no existe ningún algoritmo de resolución implementado (Warning -- It's a NLP problem. No algorithm implement), de lo contrario muestra otra advertencia anunciando que la expresión modelada se categoriza como un problema de programación lineal (Warning -- There is a LP problem, send to Garfinkell Solver) invocando la función garfinkell_solver, la cual es la encargada de realizar el llamado a la rutina que contienen los algoritmos de optimización implementados en C.*

```
class ServicesSolverAdapter
  attr_accessor :services_solver_references

  def initialize
    @services_solver_references = ServiceSolver.new
  end

  def algorithm_request expression_model                    (Código 7.10)
    puts "Paser Expression: #{expression_model}"

    #Search in expression_model if any of the equations is LP, is not, that means a NLP problem
    if (expression_model[:optimization_function] .detect {|exp| exp.is_a? Symbol})
      warn "Warning -- It's a NLP problem. No algorithm implement"
    else
      warn "Warning -- There is a LP problem, send to Garfinkell Solver"
      #garfinkell_solver is a method which connect to FFI library
       @services_solver_references.garfinkell_solver
    end

  end
end
```

Fuente: Alejandro Rodas Vásquez

Por consiguiente, se tiene una expresión matemática que ha sido analizada y convertida a una estructura de datos propia del *lenguaje anfitrión* de modo que la misma pueda ser manipulada por los componentes que conforman la arquitectura. Esto permite que el *algoritmo de resolución* obtenga en un formato estandarizado las ecuaciones que modelan el problema.

7.4.3 Integración del componente DSLException para el manejo de errores

Una de las principales funcionales que debe contar un lenguaje de programación, es la capacidad para manejar los errores que puedan ser ocasionados por el mal uso del lenguaje por parte del usuario. Al mismo tiempo, no solo se debe señalar el error cometido, sino brindar al usuario la información necesaria de modo que pueda recuperarse de su error.

Por tal motivo, se vio la necesidad de integrar a la arquitecta un componente que permitiese adaptar a las necesidades del lenguaje y que tengan como función realizar las validaciones sobre los parámetros que reciben los métodos que conformar el LEDI y en caso de un error originar el mensaje apropiado. Este componente ha sido llamado *DSLException* (código 7.13).

Sin embargo, antes es necesario describir un nuevo componente llamado *DSLConnection,* el cual ha sido construido con el propósito de ser el elemento que será invocado en el archivo (con extensión *.rb*) donde el usuario modelará su problema de optimización; este componente jugará el papel de ser la conexión entre dicho archivo y la arquitectura del LEDI.

```
require_relative '../dsl_model/optimization_model'
require_relative '../dsl_exception/dsl_exception'

module DslConnection
  def optimization_method *optimization_function_arguments
    DSLException::RestrictionException.check_keys_optimization_function(optimization_function_arguments)
    OptimizationFunction.create_object optimization_function_arguments
  end
  alias_method(:optimization_function, :optimization_method)
  alias_method(:o_f, :optimization_method)
end
```

*(C*ódigo 7*.11)*

Fuente: Alejandro Rodas Vásquez

Como se puede observar en (código 7.11), al inicio del código es necesario invocar la arquitectura del lenguaje por medio de la instrucción *require_relative '../dsl_model/optimization_model'* y al mismo tiempo el componente *DSLException* a través de la instrucción *require_relative '../dsl_exception/dsl_exception'.* Esto permite que el componente *DSLConnection* sea el punto de conexión entre la arquitecta y el archivo que manipulará el usuario.

Ahora, note como a la función *optimization_method* se le han asignado dos *alias* (como se definió en la sección *Técnicas De Metaprogramación Empleadas En El Proyecto*) nombrados como *:optimization_function* y *:o_f.* Esto mediante la instrucción *alias_method.*

Posteriormente, se tiene instrucción *DSLException::RestrictionException.check_keys_optimization_function(optimization_function_arguments),* la cual se emplea para invocar al módulo *DSLException,* donde este contiene la clase denominada *RestrictionException* que define el método *check_keys_optimization_function,* la cual es la encargada de verificar la validez de los argumentos que el usuario está introduciendo en el método *optimization_method.* De esta forma, mediante dicho método el usuario posee un interfaz que le permite hacer uso de la arquitecta que sustenta el LEDI en el archivo que él manipulará.

A continuación, se muestra el contenido del archivo nombrado como *prueba_dsl.rb* (importante *la extensión*) el cual es un ejemplo de la sintaxis resultante que ofrece el LEDI al usuario, donde este modela un problema de optimización.

```
require_relative 'builder/dsl_connection'
include DslConnection

optimization_function(:min, function_name: '3x^2+6y<=34.45').subjectTo {
    restriction_equation1 '0.03x^3>0.98'
     restriction_equation2 '10.6x<=0.002'
  }.send_result_to_solver
```

*(C*ódigo 7.*12)*

Fuente: Alejandro Rodas Vásquez

Como puede darse cuenta, al momento de iniciar el modelado del problema, es necesario invocar el componente *DSLConnection* utilizado las instrucciones *require_relative 'builder/dsl_connection'* e *include DslConnection,* respectivamente.

Ahora bien, una vez comprendida la dinámica que presenta *DSLConnection* es necesario remitirse al código mostrado en (código 7.11), enfocándose precisamente en la instrucción que hace el llamado a *DSLException* donde se utiliza el método *check_keys_optimization_function*.

Como ya se ha mencionado, *DSLException* contiene la clase *RestrictionException* la cual hereda de *Exception* (código 7.13). Esta clase es propia de *Ruby* y responsable de manejar todas aquellas excepciones que se produzcan. Como ejemplo, algunas de ellas son: *NoMemoryError, RuntimeError, SecurityError, ZeroDivisionError, y NoMethodError* [31]. Así mismo, uno de los propósitos que se busca con esta herencia es la creación de una clase diseñada exclusivamente para las necesidades del LEDI.

```
module DSLException
  class RestrictionException < Exception
    def self.validate_presence_of_restriccion(&restriccion)
      raise RestrictionException.new("The restrictions are not present") unless block_given?
    end

    def self.is_not_present?
      raise RestrictionException.new("Block is not present")
    end

    def self.is_optimization_type_present?(optimization_type)
      raise RestrictionException.new("You have to specify :min or :max") unless ([:min, :max].include?(optimization_type))
    end
```

(Código 7.13)

```
    def self.is_optimization_function_present?(arguments)
      raise RestrictionException.new("You have to specify a function like this => name_function: 'equation' ") unless
arguments.is_a?(Hash)
    end

    def self.check_number_of_arguments(optimization_function_arguments)
      raise RestrictionException.new("Wrong numbers of arguments; this is the form that you have to use [:min | :max],
name_function: 'equation' ") unless optimization_function_arguments.length == 2
    end

    def self.check_keys_optimization_function(optimization_function_arguments)
      check_number_of_arguments(optimization_function_arguments)
      is_optimization_type_present?(optimization_function_arguments[0])
      is_optimization_function_present?(optimization_function_arguments[1])
    end

  end
end
```

Fuente: Alejandro Rodas Vásquez

En (código 7.13), se puede identificar que en el cuerpo del método *check_keys_optimization_function*, se hace el llamado a tres. El primero (*check_number_of_arguments*) se encarga de verificar si la función *optimization_method* (también llamada *optimization_function* según el *alias* empleado en (código 7.12)) recibe exactamente dos argumentos. Tenga en cuenta que el primer argumento debe de ser el *tipo de optimización* de desea realizar y el segundo corresponde a la *función objetivo acompañada de su nombre.* El segundo método (*is_optimization_type_present?*) tiene como objetivo validar si el primer argumento (el *tipo de optimización*) que recibe la función *optimization_method,* es definido como *:min* o *:max.* Lo cual permite indicar si la función objetivo desea ser minimizada o maximizada.

Por último, el tercer método (*is_optimization_function_present?*) verifica si el segundo argumento que recibe la función *optimization_method,* contiene la función objetivo. Se debe considerar que para definir esta función, es necesario emplear la sintaxis que se utiliza en *Ruby* para definir una estructura de datos tipo *Hash*.

En este caso sería *name_function: 'mathematical_equation',* como se muestra en (código 7.12).

Cada uno de estos métodos crea un objeto de tipo *RestrictionException* y emplea el método *raise*[20] perteneciente a *Ruby* destinado para el manejo de excepciones. De este modo, cada vez que alguna de estas ocurra la ejecución del programa se detendrá y arrojará el mensaje respectivo.

Así pues, la adición los componentes *DSLException* y *DSLConnection* implica la modificación del *Diagrama de Componentes* propiamente en la capa *DSL Model* (Fig. 17), como también el *Diagrama de Clases* correspondiente (Fig. 18).

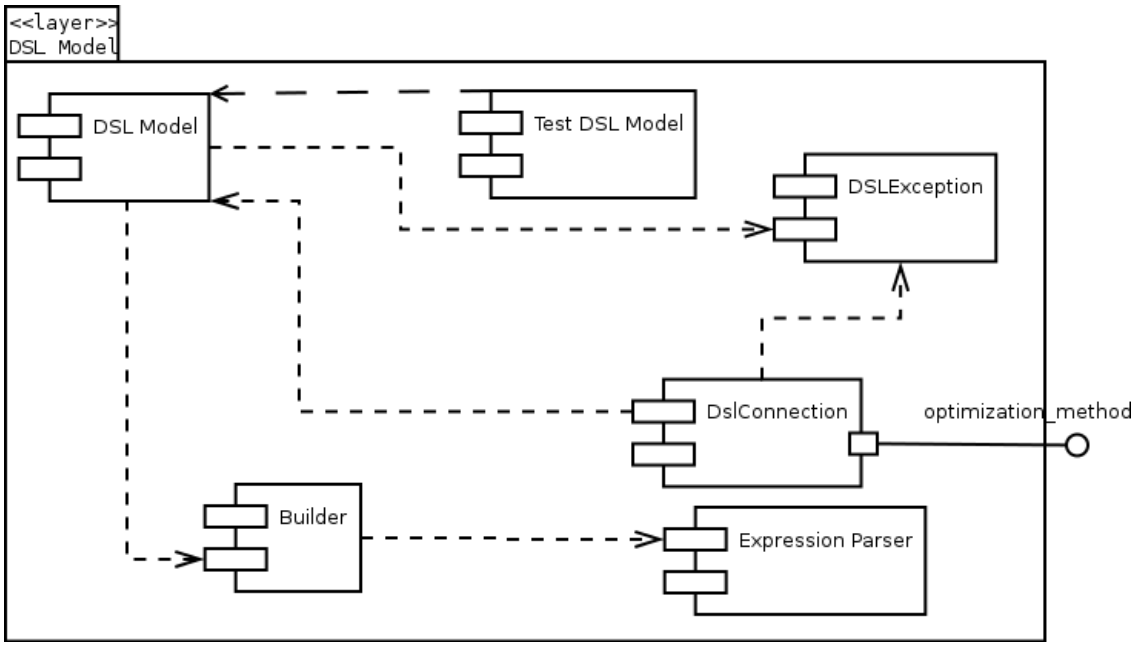


Fig. 17. Componentes *DSLException* y *DSLConnection* (Fuente: Alejandro Rodas Vásquez)

20 Cuando una excepción *emerge* (se dice que una excepción se *emerge* cuando ocurre dentro de la ejecución del programa), Ruby inmediatamente revisa su árbol de rutinas (conocida como *stack*) y busca alguna que pueda manipular la excepción en particular. Si no puede encontrar alguna rutina que pueda manejar el error existente, el programa termina con un mensaje de error

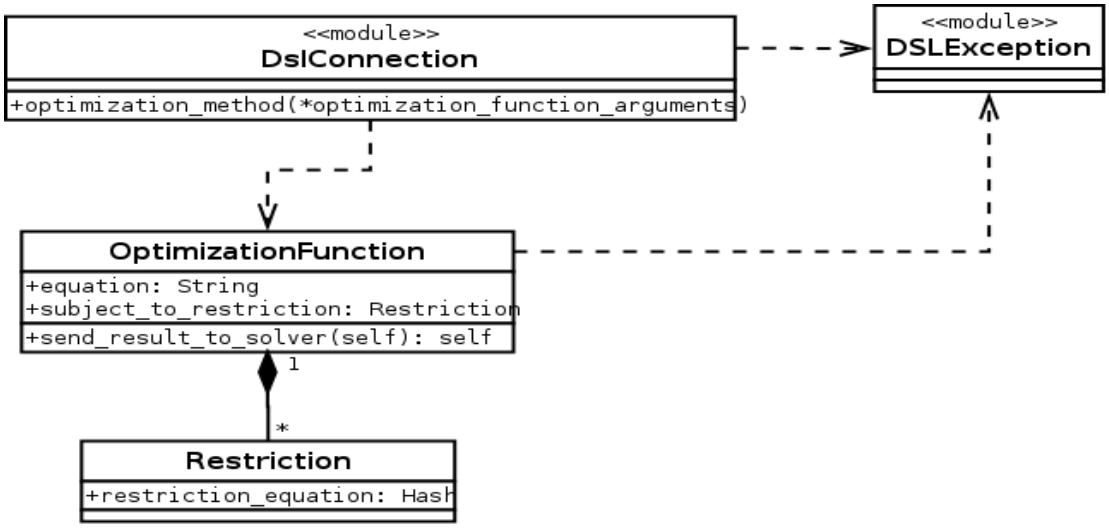


Fig. 18. Diagrama de Clases. Módulo *DSLException* (componente *DSLException*) y Módulo *DslConnection* (componente *DslConnection*) (Fuente: Alejandro Rodas Vásquez)

8. DEMOSTRACIÓN DEL FUNCIONAMIENTO DE LA ARQUITECURA MEDIANDO EL MODELADO DE UN PROBLEMA DE OPTIMIZACIÓN EMPLEANDO EL LEDI CREADO

A lo largo de este documento se muestra el proceso investigativo que culmina en la creación de un *prototipo funcional* de la arquitecta de un Lenguaje Específico de Dominio Interno orientado al modelado de problemas de optimización. Dicho proceso, se puede dividir en varias etapas; en primera instancia se presentó la metodología C4, la cual fue utilizada como marco de referencia para la construcción de arquitecta. Posteriormente, se describieron los atributos y propiedades de *Ruby* que contribuyeron en la creación de la arquitectura. Por último, se mostró el proceso evolutivo que sufrió la arquitecta desde el planteamiento que se refleja en los diagramas iniciales hasta los diagramas resultantes que se mostrarán en este apartado.

Así pues, en el presente capítulo se planteará un problema de optimización como caso práctico, el cual será modelado haciendo uso del LEDI demostrando de esta forma el funcionamiento de la arquitecta. Se utilizará el Entorno de Desarrollo Integrado *Eclipse* en cada una las pruebas.

8.1 PLANTEAMIENTO DE UN CASO PRÁCTICO EMPLEANDO EL LEDI PARA SU MODELADO

Suponga que se desea maximizar la producción de pinturas. Después de un detallado análisis el modelo resultante que permite alcanzar este objetivo es el siguiente:

Maximizar:

z=5x1+4x1

Sujeto a las restricciones:

6x1+4x2 <= 24

-2x1+2x2 <= 6

Haciendo uso del LEDI, este modelo se puede plantear de la siguiente forma:

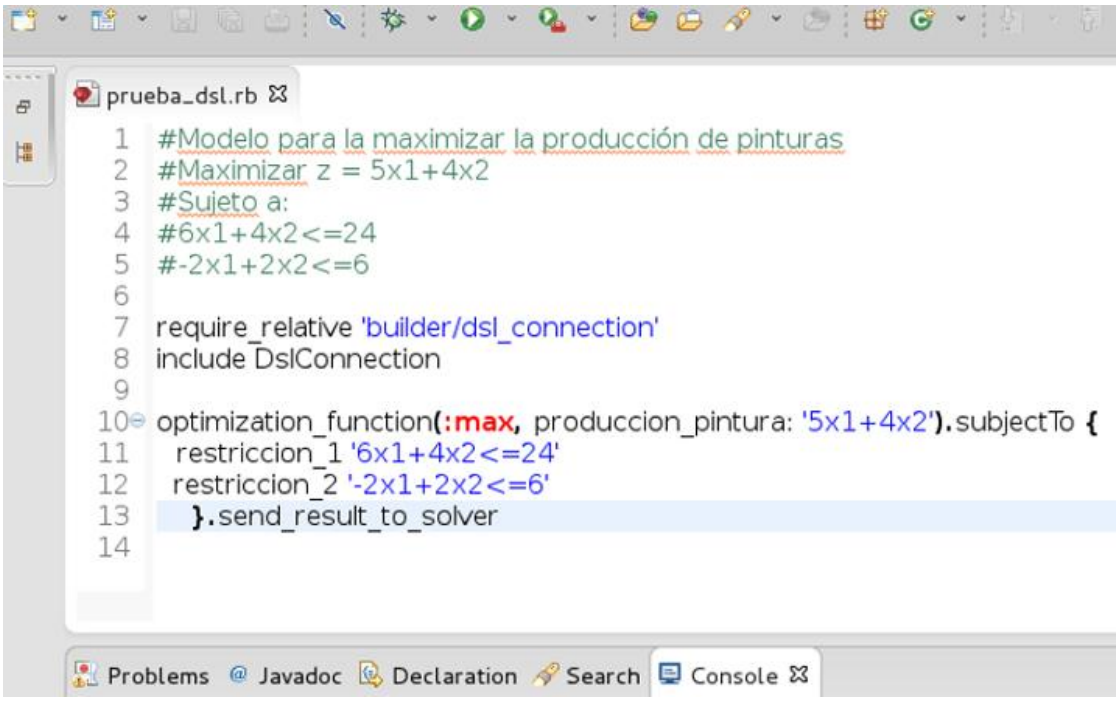


Fig. 19. Problema de optimización modelado a través del LEDI utilizando el *Ambiente de Desarrollo Integrado Eclipse*.(Fuente: Alejandro Rodas Vásquez)

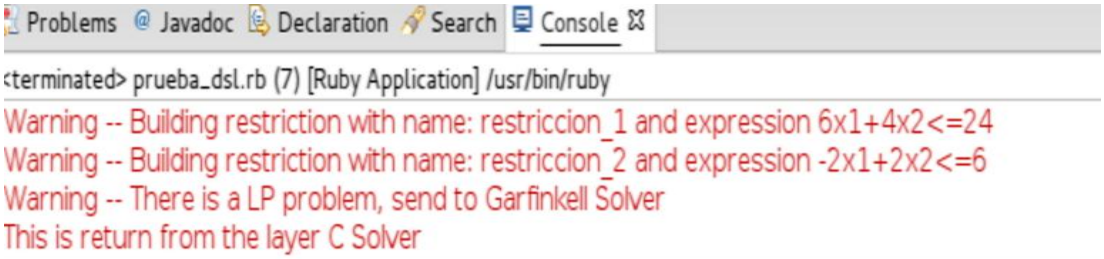


Fig. 20. Salida por consola originada por el LEDI (Fuente: Alejandro Rodas Vásquez)

Note que antes de iniciar el modelado del problema, se debe invocar la librería y el módulo de conexión para hacer uso del lenguaje, tal como se puede observar en las líneas 7 y 8 de la Fig. 19.

Para iniciar el modelado es *necesario* utilizar la función *optimization_function*, la cual también puede ser invocada por su *alias o_f*, como se mencionó en apartados anteriores. Esta función acepta dos parámetros, el primero es el tipo de optimización que se desea realizar, ya sea maximizar o minimizar. Esto es necesario especificarlo mediante los indicadores *:max o :min.* En el segundo parámetro se especifica la función de optimización con un nombre que la caracterice, como se puede notar en Fig. 19 línea 10. Posteriormente, se definen las restricciones a las que está sujeta la función de optimización, esto se realiza mediante la función *subjectTo* (la cual también puede ser invocada como *s_t*)

Del mismo modo, en la línea 11 y 12 de Fig. 19 se especifican las restricciones del modelo. Note que a cada una de ellas se le ha proporcionado un nombre, esto con el fin de identificarlas. Por último, la función *send_result_to_solver* envía el modelo hacia los algoritmos de resolución, generando como resultado la salida por consola que se observan en la Fig. 20. Los mensaje que se muestran fueron diseñados con el fin de visualizar el proceso que ejecuta la arquitectura, de modo que el usuario este informado.

Los dos primeros mensajes (*Warning -- Building restricction with name: restricccion_1 and expression 6x1+4x2<=24* y *Warning -- Building restriction with name: restriccion_2 and expression -2x1+2x2<=6*) son generados por el módulo *RestrictionBuilder* al emplear la técnica de *Definición dinámica de métodos* (como se explicó anteriormente en el apartado *Técnicas De Metaprogramación Empleadas En El Proyecto*), esto con propósito que el usuario pueda saber que la arquitectura está construyendo las restricciones del modelo y así una vez verificada la sintaxis de las expresiones matemáticas, la arquitectura procederá a determinar si el modelo cabe dentro de la categoría de *Programación Lineal o Programación No Lineal.* Para este caso la arquitectura ha determinado que el modelo es de tipo *Programación Lineal* y será resuelto por el *algoritmo de resolución* aplicando el método de *Garfinkell*, como es informado por el mensaje *Warning -- There is a LP problem, send to Garfinkell Solver* (Fig. 20). Cabe aclarar que dentro del alcance del proyecto no se contempló la implementación de dicho algoritmo, dejándolo así para trabajos futuros.

Por último, se observa el siguiente mensaje: *This is return from the layer C Solver.* De esta manera se demuestra que la petición realizada a través de la arquitectura ha alcanzado la capa *C Solver*, llamando así el algoritmo implementado en el lenguaje C (Fig. 21).

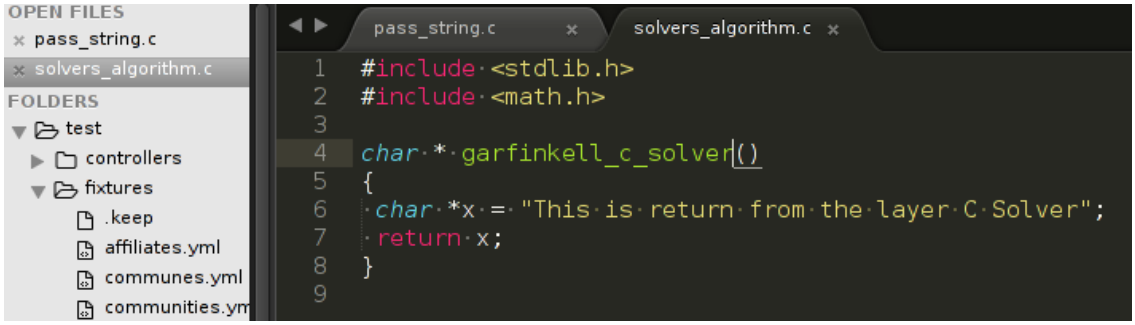


Fig. 21. Implementación del método *garfinkell_c_solver* empleando el lenguaje de programación C en el editor *SublimeText*. Fuente: Alejandro Rodas Vásquez

Ahora, para demostrar que la arquitectura está en capacidad para reconocer un problema de optimización que no pertenece a la categoría de *Programación Lineal*, se plantea en la Fig. 22 un problema con estas características.

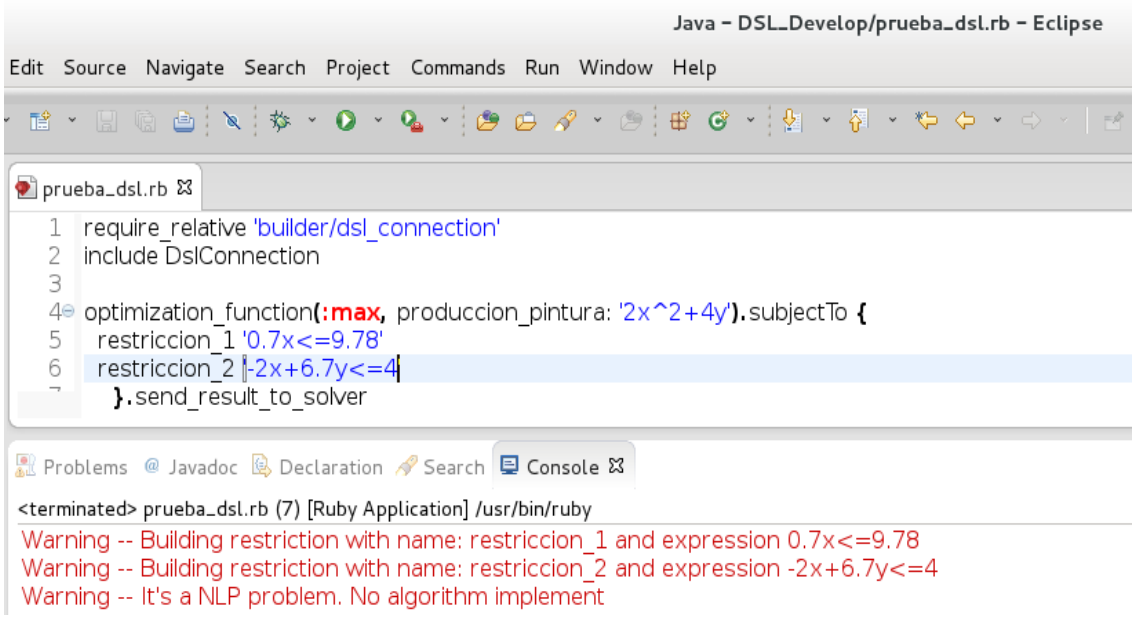


Fig. 22. Modelado de un problema de optimización en la categoría de *Programación No Lineal* (Fuente: Alejandro Rodas Vásquez)

Observe que la función a optimizar se describe como es *2x^2+4y,* donde la variable *x* es elevada al cuadrado, sin embargo la arquitectura está en capacidad de detectarlo, como se muestra en la sección inferior de la imagen (vista por consola) mediante el mensaje *Warning -- It's a NLP problem. No algorithm implement*. En la Fig. 23 se muestra el mismo modelo que la figura anterior pero haciendo uso de los *alias o_f* y *s_t.*

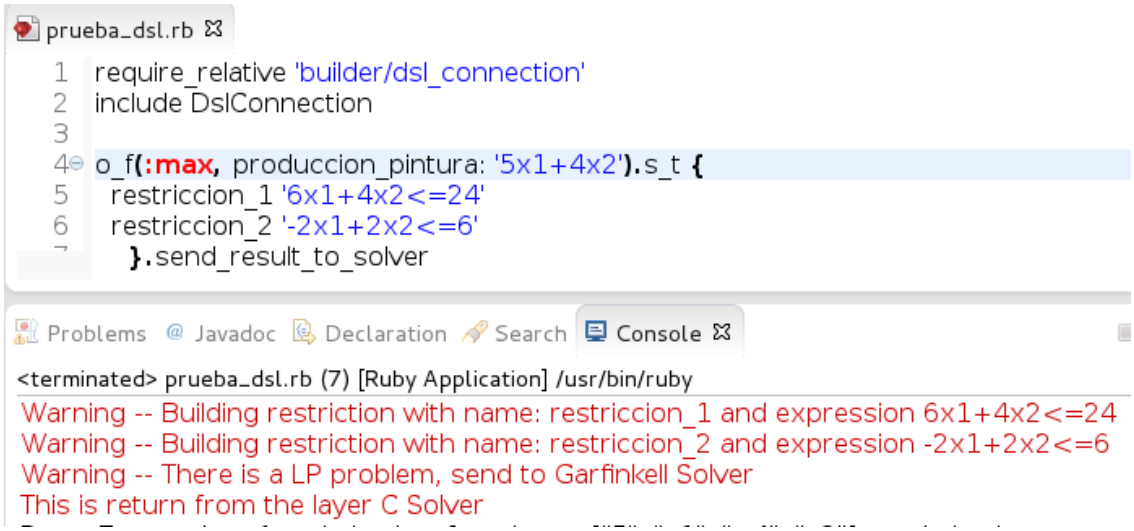


Fig. 23. Modelado de un problema de optimización en la categoría de *Programación No Lineal,* empleando los *alias o_f* y *s_t.* (Fuente: Alejandro Rodas Vásquez)

Uno de los principales atributos que debe contar un lenguaje es su capacidad para detectar errores originados por el usuario al realizar un mal uso de la sintaxis del mismo, por lo tanto este *prototipo funcional* de la arquitecta cuenta con el módulo *DSLException* para realizar dicha función, para observar el funcionamiento de este módulo se plantea el modelado de un problema de optimización pero esta vez omitiendo parámetros que son obligatorios para los métodos que componen el LEDI.

Suponga que usted tiene el mismo problema que ha sido modelado en Fig. 23, pero esta vez ha olvidado especificar el tipo de optimización, es decir, el parámetro :*max* no ha sido proporcionado (Fig. 24), la arquitectura está en condiciones de verificar el número de parámetros que debe recibir el método *o_f* y emitir el error respectivo.

```
require_relative 'builder/dsl_connection'
include DslConnection

o_f( produccion_pintura: '5x1+4x2').s_t {
  restriccion_1 '6x1+4x2<=24'
  restriccion_2 '-2x1+2x2<=6'
    }.send_result_to_solver
```

```
Wrong numbers of arguments; this is the form that you have to use [:min | :max], name_function: 'equation'
```

Fig. 24. Error: números equivocados de parámetros (Fuente: Alejandro Rodas Vásquez)

Como puede observar en la Fig. 24, no solo se indica el error, sino que el mensaje también ofrece el formato correcto en que se deben presentar los parámetros que debe recibir el método. Por otro lado, en Fig. 25 aunque se presenta el mismo modelo del problema de optimización, en esta ocasión se ha introducido como primer parámetro de la función *o_f*, la palabra *:aux* (línea 4) lo cual es obviamente un error, pues como se puede recordar este parámetro solo puede recibir las palabras reservadas *:min* o *:max*, por lo tanto se debe generar un error, tal como se muestra en la sección inferior de la figura

```
require_relative 'builder/dsl_connection'
include DslConnection

o_f(:aux, produccion_pintura: '5x1+4x2').s_t {
  restriccion_1 '6x1+4x2<=24'
  restriccion_2 '-2x1+2x2<=6'
    }.send_result_to_solver
```

```
`is_optimization_type_present?': You have to specify :min or :max (DSLException::RestrictionException)
```

Figura 25. Error: no se debe especificar en el argumento *tipo de optimización* valores distintos a *:min o :max* (Fuente: Alejandro Rodas Vásquez)

Así mismo, un error que puede ocurrir frecuentemente radica en la forma de definir la función de optimización. Recuerde que el formato apropiado para esto es la siguiente:

nombre_de_funcion: 'expresión_matemática'

Es importante enfatizar que la ecuación o expresión que define la función de optimización debe ir entre comillas dobles (“) o simples ('). Como se puede observar en la Fig. 26 línea 4, se ha introducido la función a optimizar en un formato erróneo, de modo que la arquitectura al evaluar dicha expresión emite el error que se puede observar en la parte inferior de la figura. Note que el mismo mensaje de error guía al usuario mostrándole cómo debe de ser el formato correcto.

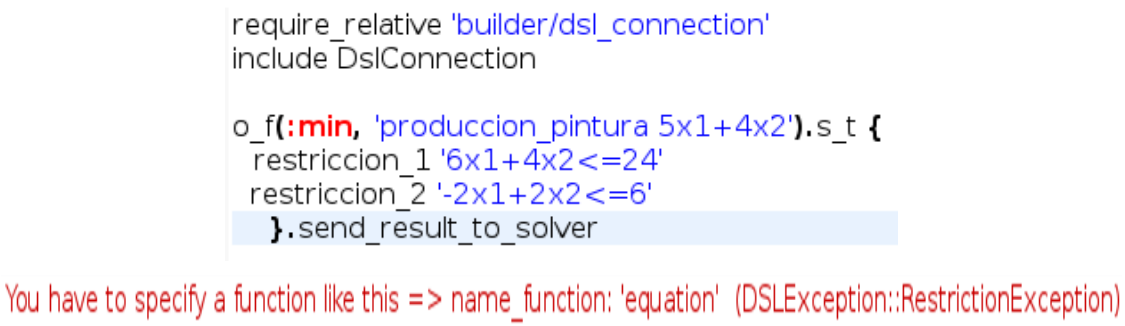


Fig 26. Error: el segundo parámetro del método *o_f* debe presentar el formato nombre_funcion: 'ecuacion' (Fuente: Alejandro Rodas Vásquez)

Por otro lado, puede existir el caso donde el usuario olvide especificar las restricciones del modelo a optimizar, tal como se muestra en la Fig 27. Recuerde que después de definir el tipo de optimización que se desea y la expresión matemática a optimizar, la función *s_t* o *subjectTo* debe de recibir las restricciones pertinentes. Así pues, la arquitectura por medio de su módulo *DSLException*, está en capacidad para detectar si no se ha especificado algún tipo de restricción, originando el mensaje de error como se puede notar en la figura.

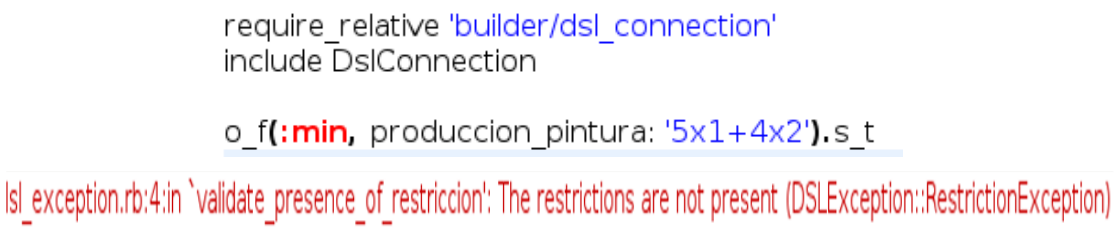


Fig. 27. Error: se debe especificar las restricciones a las que está sujeta la función de optimización (Fuente: Alejandro Rodas Vásquez)

Para terminar, los casos propuestos en este capítulo se construyeron con el fin de exponer el funcionamiento de la arquitecta que sustenta el LEDI, la cual fue construida mediante un proceso meticuloso de experimentación y mejoramiento continuo con el fin de brindarle al usuario un lenguaje adaptado al *dominio-específico* enfocado en el área de la optimización. Por tanto, la arquitectura fue sufriendo cambios desde su etapa inicial, teniendo como objetivo obtener un producto que tuviera como atributos de calidad propios de una arquitecta de software, la capacidad de *modificabilidad* e *integración* [38].

En los Anexos B y C se muestran como producto final los *Diagramas de Componentes* y *Clase*, respectivamente, los cuales son resultados finales de la investigación realizada.

CONCLUSIONES

1. Durante el proceso investigativo del proyecto se pudo constatar que el campo referente a la creación de Lenguajes Específicos de Dominio Interno ha sido explorado de diferentes formar, siendo el desarrollo web un nicho importante. En esta área se encontraron LEDI como Ruby on Rails y Rspec, el cual ha tomado fuerza en el sector de testing y pruebas de integración.

2. Al iniciar la construcción de un software se requiere seleccionar alguna metodología. Sin embargo, al desarrollar el LEDI se necesitaba que la misma se concentrara específicamente en la captura del conocimiento del dominio y que dicho conocimiento pudiera ser transferido e implementado a través de código fuente. En tal escenario, el Diseño Basado en Modelos (Model Driven Desing) fue una herramienta de gran valor ya que permitió llevar la construcción hacia tal enfoque, puesto que el eje central del desarrollo era el Modelo en sí.

3. Como se puede constatar en el texto, UML ofrece una variedad de diagramas que permiten observar el software o una aplicación determinada desde varias Vistas. Sin embargo, al iniciar el planteamiento de la arquitectura del LEDI se encontró que esta diversidad de diagramas en realidad planteaba cierta desventaja puesto que no se tenía en claro cuál de todos seleccionar. En este punto era importante encontrar alguna metodología que brindara un punto de referencia. Es allí donde la metodología C4 ofrece dentro de su planteamiento una forma práctica y sencilla para construir un sistema, esto permitió crear la arquitectura con un enfoque funcional e incremental, ya como se puede evidenciar en el documento, a medida que se fue necesitando la incorporación de nuevos componentes que desempeñaran distintas funcionalidades, estos fueron acoplados sin problema en la arquitectura.

4. Durante las pruebas que se realizaron en el capítulo 9, se evidenció que la arquitectura se comporta de la manera esperada, presentando un funcionamiento correcto a la hora de verificar el empleo de la sintaxis. Del mismo modo, el componente DSLException juega el papel planeado ya que no solo captura las excepciones generadas sino que los que mensajes que muestra son descriptivos para el usuario, tal y como originalmente fue concebido dicho componente.

BIBLIOGRAFÍA

[1] Juan de Lara and Esther. Guerra. “Domain-Specific Textual Meta-Modelling Languages for Model Driven Engineering”, Modelling Foundations and Applications, pp. 259-274, 2012

[2] Janne Luoma, Steven Kelly and Juha-Pekka Tolvanen. “Defining Domain-Specific Modeling Languages Collected Experiences”, in Conf SPLC'05 Proceedings of the 9th international conference on Software Product Lines, pp 198 – 209.

[3] ZSCHALER, Steffen. KOLOVOS, Dimitrios S. DRIVALOS, Nikolaos. PAIGE, Richard F. y RASHID, Awais . Domain-Specific Metamodelling Languages for Software Language Engineering. [en línea]. Disponible en Internet: http://steffen-zschaler.de/publications/sle09_dsm2ls.pdf

[4] GHOSH, Debasish. DSLs in Action. Stamford: Manning, 2010.

[5] CZARNECKI, Krzysztof. Overview of Generative Software Development. En: Unconventional Programming Paradigms. Saint-Michel: Springer-Verlag, 2004, p. 326-341.

[6] FOWLER, Martin. Domain – Specific Languages. Boston: Addison Wesley, 2011.

[7] NEIGHBORS, James. M. The Evolution from Software Components to Domain Analysis. [en línea]. Disponible en Internet: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.21.6007&rep=rep1&type=pdf

[8] IRAZÁBAL, Jerónimo. PÉREZ, Gabriela. PONS, Claudia. y GIANDINI Roxana. An implementation approach to achieve MetaModel independence in Domain Specific Model Manipulation Languages. [en línea]. Disponible en Internet: http://imgbiblio.vaneduc.edu.ar/fulltext/files/TC105242.pdf

[9] BECK, Howard. CURRIE, Ken y TATE, Austin. A Domain Description Language for Job-Shop Scheduling. En: Artificial Intelligence Applications Institute. [en línea]. Disponible en Internet: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.35.269&rep=rep1&type=pdf

[10] KLINT, Paul. y VAN DEURSEN, Arie. Little Languages: Little Maintenance?. En: Journal of Software Maintenance: Research and Practice. No 10. (Abril. 1998). p. 75-92 [en línea]. Disponible en Internet: http://homepages.cwi.nl/~paulk/publications/JSM98.pdf

[11] GÜNTHER, Sebastian. Agile DSL-Engineering with Patterns in Ruby. 2009

[12] VOELTER, Markus. DSL Engineering. Designing, Implementing and Using Domain-Specific Languages. [en línea]. Disponible en Internet: http://dslbook.org

[13] EVANS, Eric. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley, 2003. ISBN 0-321-12521-5

[14] Robert Fourer, David M. Gay, Brian W. Kernighan, AMPL A Modeling Language for Mathematical Programming, 2da ed. 2003.

[15] IBM ILOG OPL Language User's Manual [en línea]. Disponible en Internet: http://cedric.cnam.fr/~lamberta/MPRO/ECMA/doc/oplTutorial.pdf

[16] FOURER, Robert. Algebraic Modeling Languages for Optimization. [en línea]. Disponible en Internet: http://ampl.com/REFS/amlopt.pdf

[17] MERNIK, Marjan. When and How to Develop Domain-Specific Languages. En: ACM Computing Surveys. No 37. (Dic 2005). p 316-344.

[18] VAN DEURSEN, Arie. KLINT, Paul y VISSER, Jooster. Domain-Specific Languages: An Annotated Bibliography. [en línea]. Disponible en Internet: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.8207

[19] GÄRTNER, Johannes. MUSLIU, Nysret. SCHAFHAUSER, Wener. y SLANY, Wolfgang. A Domain Specific Language for Modeling and Solving Staff Scheduling Problems. [en línea]. Disponible en Internet: http://www.dbai.tuwien.ac.at/staff/musliu/CischedEMPLE.pdf

[20] MEDIRATTA, Anupam. A Generic Domain Specific Language For Financial Contracts. 2007. Tesis (Master of Science). The State University of New Jersey.

[21] HALPIN, Terry. UML Data Models From An ORM Perspective: Part 1.[en línea]. Disponible en Internet: http://www.orm.net/pdf/ICMArticle1.pdf

[22] FLANAGAN, David. y MATSUMOTO, Yukihiro. The Ruby Programming Language. California: O'Reilly, 2008. ISBN-10 0-596-51617-7

[23] FIRESMITH, Donald G. Object-Oriented Requirements Analysis and Logical Design. Wiley, 1993. ISBN-13: 978-0471578079

[24] PRESSMAN, Roger. S. Ingeniería del Software: Un enfoque práctico. 7. ed. New York: McGraw-Hill, 2010.

[25] BECK, Kent. Implementation Patterns. 7. ed. Boston: Addison-Wesley Professional, 2007.

[26] RUMBAUGH, James. JACOBSON Ivar. y BOOCH. Grady. El Lenguaje Unificado de Modelado: Manual de Referencia. Madrid: Addison-Wesley, 2000. ISBN: 8478290370

[27] BROWN, Simon Software Architecture for Developers. Leanpub, 2015

[28] RUBIN, Jeff. y CHISNELL, Dana. Handbook of Usability Testing, 2 ed. Indianapolis: Wiley, 2008. ISBN 978-0-470-18548-3

[29] International Standard ISO/IEC 9126-1. [en línea]. Disponible en Internet: http://www.cse.unsw.edu.au/~cs3710/PMmaterials/Resources/9126-1%20Standard.pdf.

[30] ISO 9241-11 Ergonomic requirements for office work with visual display terminals [en línea]. Disponible en Internet: http://www.iso.org/iso/catalogue_detail.htm?csnumber=16883

[31] COOPER, Peter. Beginning Ruby From Novice to Professional, 2. ed. New York: Apress, 2009. ISBN-13 (pbk) 978-1-4302-2363-4

[32] CARLSON, Lucas y RICHARDSON, Leonard. Ruby Cookbook. Sebastopol: O'Reilly, 2006. ISBN 0-596-52369-6

[33] MARTIN, Robert C, Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2008. ISBN-10 0132350882

[34] Paolo Perrotta, Metaprogramming Ruby. The Pragmatic Programmers; 2010

[35] El proceso de compilación, del código fuente al código máquina [en línea] Disponible en Internet: http://www.uhu.es/04004/material/Transparencias3.pdf

[36] Administración de Memoria: Memoria principal [en línea] Disponible en Internet: http://www.fing.edu.uy/tecnoinf/mvd/cursos/so/material/teo/so08-memoria.pdf

[37] GRAMMA, Eric. HELM, Richard, JHONSON, Ralph. y VLISSIDES, Jhon. Design Patterns: Elements of Reusable Object-Oriented Software. Massachussets: Addison Wesley, 2014. ISBN 0-201-63361-2

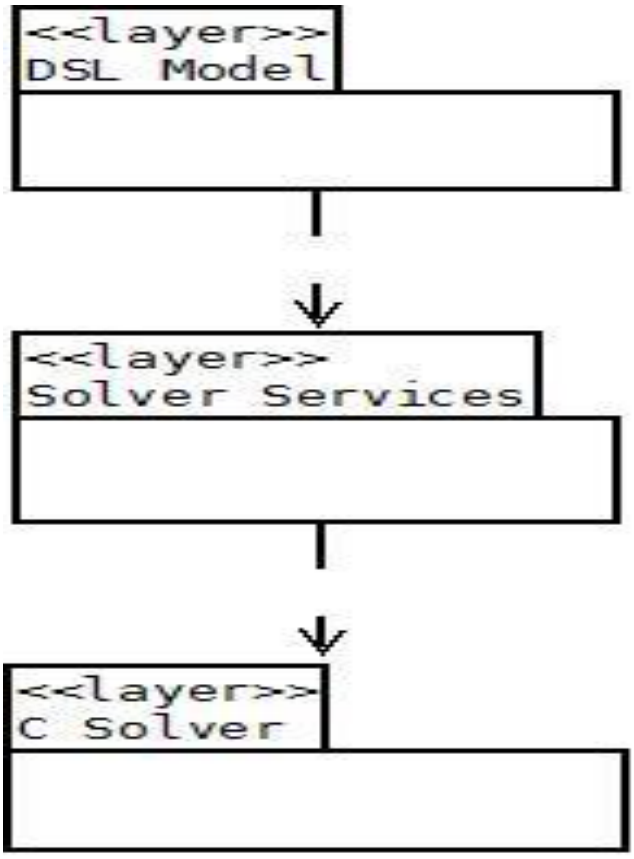
[38] GORTON, Ian. Essential Software Architecture. New York: Springer, 2006. ISBN-10 3-540-28713-2

[39] Truls Flatberg. A short OPL Tutorial [en línea] Disponible en Internet: http://folk.uio.no/trulsf/opl/opl_tutorial.pdf

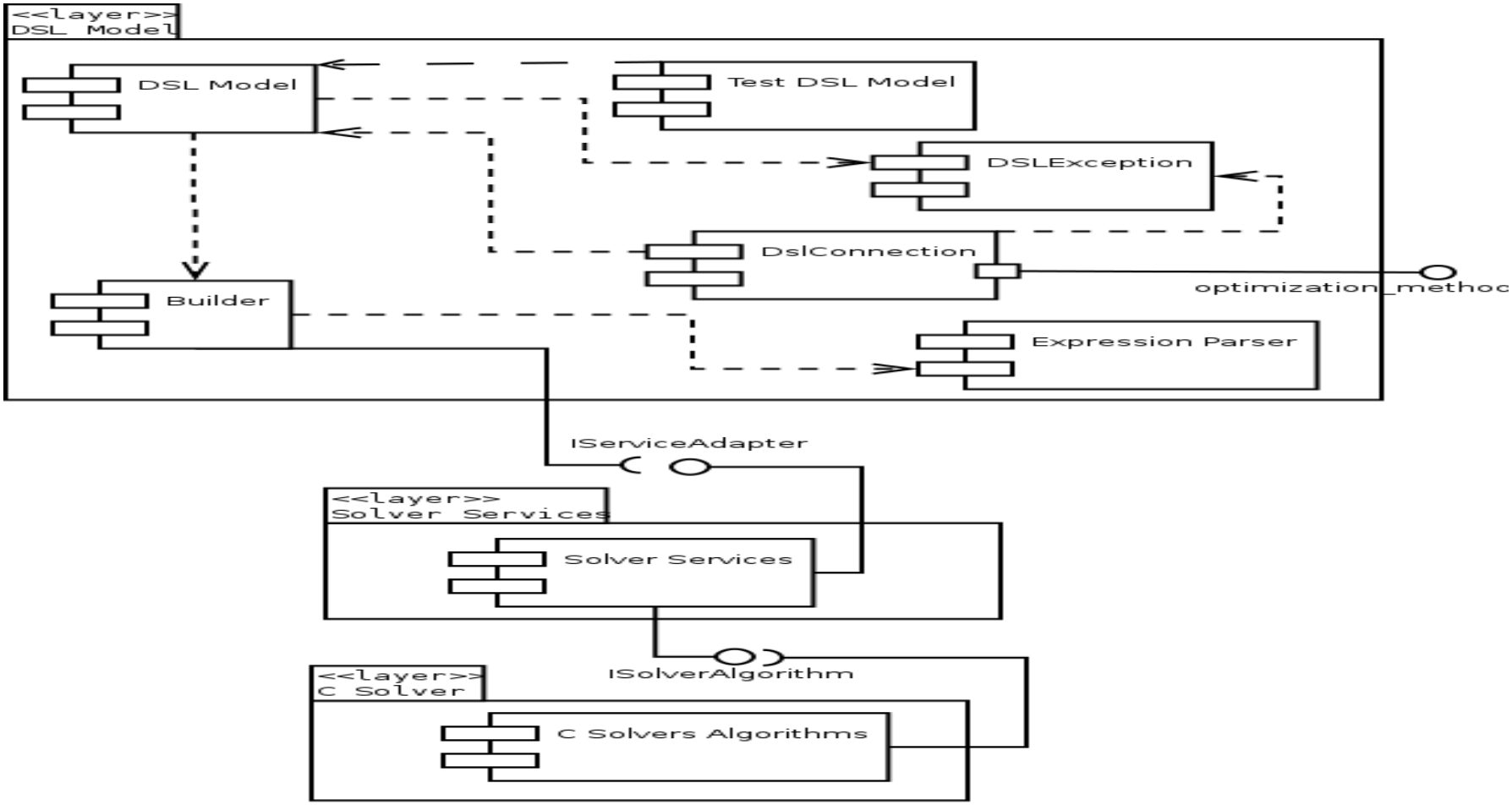
[40] BROWN, S.A, DRAYTO, C.E y MITTMAN, B. A. Description of the APT Language. En: Communications of the ACM. No.6. (Nov. 1963). p. 649-658. [en línea] Disponible en Internet: http://glennbonner.com/pdfs/apt-description.pdf

[41] Paul Hudak. Domain Specific Languages. Department of Computer Science Yale University. [en línea] Disponible en Internet: http://haskell.cs.yale.edu/wp-content/uploads/2011/01/DSEL-Little.pdf

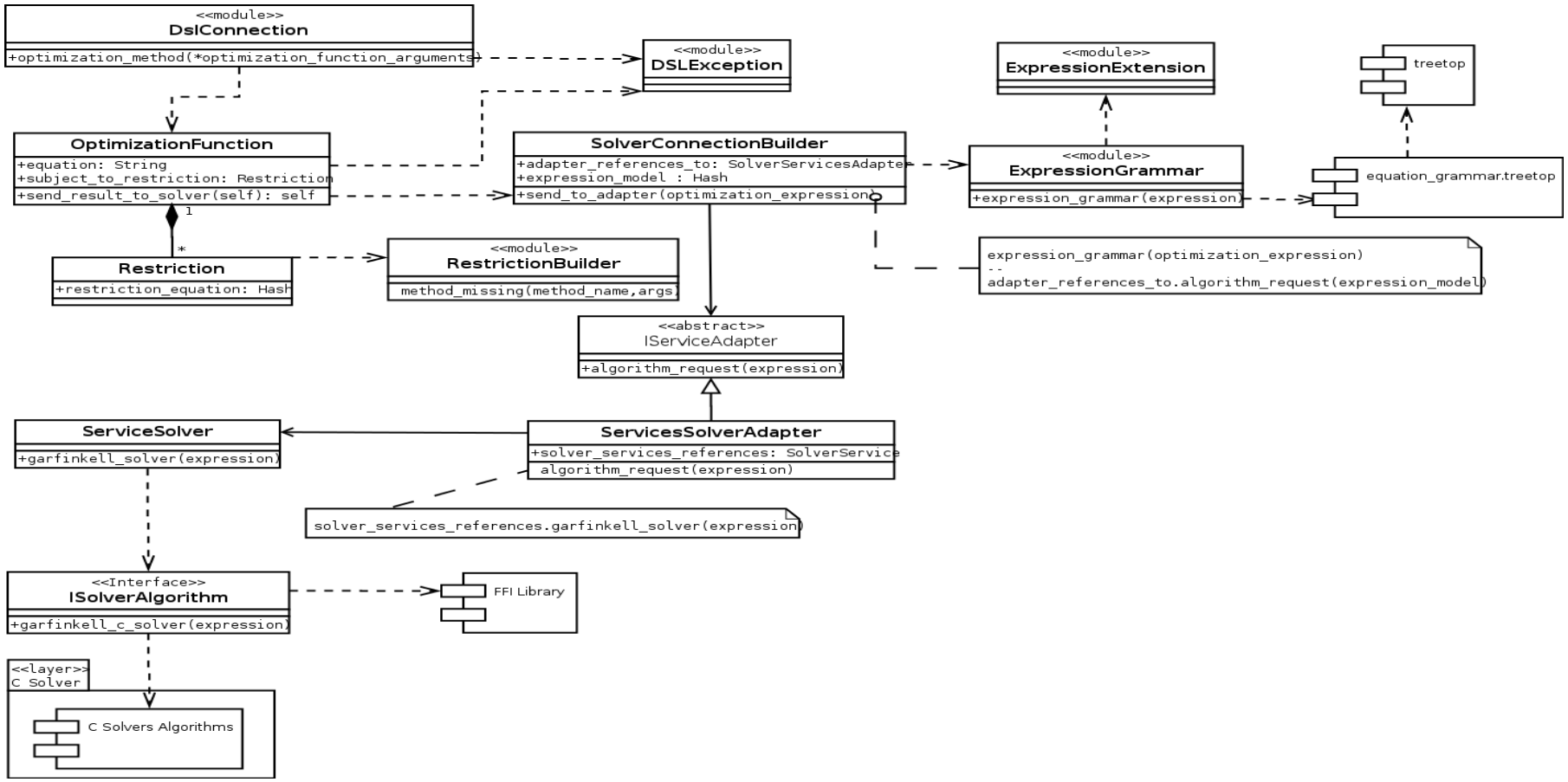
ANEXO A. Arquitectura por capas del LEDI



ANEXO B. Diagrama de Componentes Final



ANEXO C. Diagrama de Clases Final



ANEXO D. dsl_connection.rb

```
require_relative '../dsl_model/optimization_model'
require_relative '../dsl_exception/dsl_exception'

module DslConnection
  def optimization_method *optimization_function_arguments

DSLException::RestrictionException.check_keys_optimization_function(optimizatio
n_function_arguments)
    OptimizationFunction.create_object optimization_function_arguments
  end
  alias_method(:optimization_function, :optimization_method)
  alias_method(:o_f, :optimization_method)
end
```

ANEXO E. optimization_model.rb

```
require_relative '../builder/restriction_builder'
require_relative '../builder/solver_connection_builder'
class OptimizationFunction
  attr_accessor :equation, :subject_to_restriction, :optimization_type
  #include ConstantFunction
  def initialize equation, optimization_type
    @equation = equation
    @optimization_type = optimization_type
  end

  def self.create_object equation
    optimization_type = equation[0]
    defined_equation = equation[1].values[0]
    funcion_optimizacion = new defined_equation, optimization_type
    funcion_optimizacion
  end

  def subject_to &block
    restriction = Restriction.new
    restriction.instance_eval(&block)
    @subject_to_restriction = restriction
    self
  end
  alias_method(:subjectTo, :subject_to)
  alias_method(:s_t, :subject_to)

  def send_result_to_solver
     SolverConnectionBuilder.new.send_to_adapter(self)
    self
  end

  class Restriction
    attr_accessor :restriction_equation
    include RestrictionBuilder
    def initialize
      @restriction_equation = {}
    end

    def restriction_number?
      @restriction_equation.size
    end
  end

end
```

ANEXO F. restriction_builder.rb

```
module RestrictionBuilder
  def method_missing(method_name, args)
    eigenclass = class << self; self; end

    eigenclass.class_eval do
      define_method(method_name) do
        # Build Hash like {'restriction_name' => restriction_equation}
        warn "Warning -- Building restriction with name: #{method_name} and
expression #{args}"
        @restriction_equation[method_name.to_sym] = args
      end

    end
      send(method_name)

  end

end
```

ANEXO G. dsl_exception.rb

```
module DSLException
  class RestrictionException < Exception
    def self.validate_presence_of_restriccion(&restriccion)
      raise RestrictionException.new("The restrictions are not present") if
(restriccion.call.nil?) && (block_given?)
    end

    def self.is_not_present?
      raise RestrictionException.new("Block is not present")
    end

    def self.is_optimization_type_present?(optimization_type)
      raise RestrictionException.new("You have to specify :min or :max") unless
([:min, :max].include?(optimization_type))
    end

    def self.is_optimization_function_present?(arguments)
      raise RestrictionException.new("You have to specify a function like this =>
name_function: 'equation' '") unless arguments.is_a?(Hash)
    end

    def self.check_number_of_arguments(optimization_function_arguments)
      raise RestrictionException.new("Wrong numbers of arguments; this is the form
that you have to use [:min | :max], name_function: 'equation' '") unless
optimization_function_arguments.length == 2
    end

    def self.check_keys_optimization_function(optimization_function_arguments)
      check_number_of_arguments(optimization_function_arguments)
      is_optimization_type_present?(optimization_function_arguments[0])
      is_optimization_function_present?(optimization_function_arguments[1])
    end

  end

end
```

ANEXO H. solver_connection_builder.rb

```
require_relative '../services_solver/services_solver_adapter'
require_relative '../expression_parser/dsl_grammar'
class SolverConnectionBuilder

  #Include a expression parser module. Implement in TreeTop
  include ExpressionGrammar
  def send_to_adapter optimization_expression

    adapter_references_to = ServicesSolverAdapter.new

    #This array contains the Optimization Function and Restrictions once these are
parser
    expression_model = {}
    expression_model[:optimization_function] =
expression_grammar(optimization_expression.equation)
    expression_model[:optimization_type] =
optimization_expression.optimization_type

    optimization_expression.subject_to_restriction.restriction_equation.each do
|key, value|
      expression_model[key.to_sym] = expression_grammar(value)
    end

    adapter_references_to.algorithm_request(expression_model)
  end
end
```

ANEXO I. services_solver_adapter.rb

```
require_relative 'service_solver'

class ServicesSolverAdapter
  attr_accessor :services_solver_references

  def initialize
    @services_solver_references = ServiceSolver.new
  end

  def algorithm_request expression_model
    puts "Paser Expression: #{expression_model}"

    #Search in expression_model if any of the equations is LP, is not, that means a
NLP problem
    if (expression_model[:optimization_function] .detect {|exp| exp.is_a? Symbol})
      warn "Warning -- It's a NLP problem. No algorithm implement"
    else
      warn "Warning -- There is a LP problem, send to Garfinkell Solver"
      #garfinkell_solver is a method which connect to FFI library
       @services_solver_references.garfinkell_solver
    end

  end
end
```

ANEXO J. service_solver.rb

```
require 'ffi'

module ISolverAlgorithm
  extend FFI::Library

  #ruta del directorio donde estan los algoritmos en C de optimizacion
  ffi_lib 'c_solvers/solvers_algorithm.so'
  attach_function :garfinkell_c_solver,[],:int
  #attach_function :garfinkell_c_solver,[],:string
end

class ServiceSolver
  include ISolverAlgorithm

  def garfinkell_solver
    garfinkell_c_solver
  end
end
```

ANEXO K. expression_grammar.rb

```
require 'rubygems'
require 'treetop'

require_relative  'expression_extensions'
base_path = File.expand_path(File.dirname(__FILE__))
Treetop.load(File.join(base_path, 'equation_grammar.treetop'))

module ExpressionGrammar
  def expression_grammar(expression)
    parser = ExpressionGrammarParser.new
    parser.parse(expression).content
  end
end
```

ANEXO L. expression_extension.rb

```
module ExpressionExtension
  class EquationLiteral < Treetop::Runtime::SyntaxNode
    def content
      elements.map { |node| node.content   unless node.class.name ==
"Treetop::Runtime::SyntaxNode" }
    end
  end

  class ExpressionLiteral < Treetop::Runtime::SyntaxNode
    def content
      self.text_value
    end
  end

  class NumberLiteral < Treetop::Runtime::SyntaxNode
    def content
      self.text_value
    end
  end

  class IdentifierLiteral < Treetop::Runtime::SyntaxNode
    def content
      ["^"] == self.text_value.scan(/\^/)) ? self.text_value.to_sym : self.text_value
    end
  end

  class OperadorLiteral < Treetop::Runtime::SyntaxNode
    def content
      self.text_value
    end
  end

end
```

ANEXO M. equation_grammar.treetop

```
grammar ExpressionGrammar

rule equation
   (number /  identifier / equality_inequality)*
<ExpressionExtension::EquationLiteral>
  end


  rule expression
   number identifier? <ExpressionExtension::ExpressionLiteral>
  end

  rule number
   ('+' / '-')?  [0-9]+ ('.' [0-9]+)? <ExpressionExtension::NumberLiteral>
  end

  rule identifier
    [a-zA-Z]+ ([0-9]+)? ('^' [0-9]+)? <ExpressionExtension::IdentifierLiteral>
  end

   rule equality_inequality
    ( '=' /  '>='  / '<=' / '<' / '>')+  <ExpressionExtension::OperadorLiteral>
   end

end
```